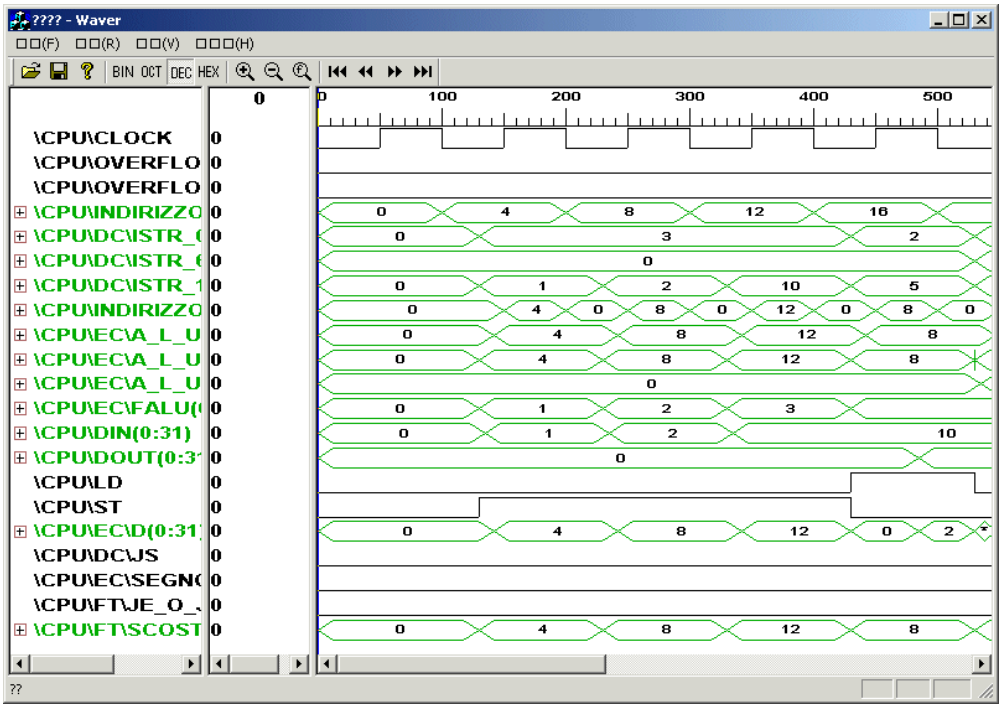
UNIVERSITA' DEGLI STUDI DI FIRENZE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Elaborato per l'esame di
"Calcolatori Elettronici" A.A.1999-2000
Prof. G. Bucci

Andrea Fedeli Sauro Menchetti

Realizzazione in VHDL e simulazione di tre CPU



Sommario

INTRODUZIONE 2

1.1 UN ESEMPIO 3

PRINCIPALI CARATTERISTICHE DEL LINGUAGGIO 5

2.1 ENTITÀ ED ARCHITETTURE 5
2.1.1 Dichiarazione di un'Entità 6
2.1.2 Dichiarazione e corpo di un'Architettura 7
2.2 TIPI DI DATO 7
2.3 UNITÀ DI PROGETTO 8
2.3.1 Entità 9
2.3.2 Architetture 9
2.3.3 Pacchetti e corpo dei pacchetti 9
2.3.4 Configurazioni 10
2.4 LIVELLI DI ASTRAZIONE 11
2.5 L'ISTRUZIONE `PROCESS` 12
2.5.1 L'istruzione `process` senza lista di sensitività 14
2.6 VHDL CONCORRENTE E SEQUENZIALE 15
2.7 TEST BENCH 16

SVILUPPO DI CPU1 18

3.1 FILOSOFIA DI PROGETTAZIONE 18
3.2 MODELLAZIONE DEI PRINCIPALI COMPONENTI 19
3.2.1 I nomi dei file 19
3.2.2 Il pacchetto `cpu_tipi` 20
3.2.3 Il register file 20
3.2.4 La ALU 21
3.2.5 La memoria 22
3.3 CPU1 DESCRITTA NEGLI APPUNTI 23
3.3.1 L'errore 24
3.4 CPU1 CORRETTA 25
3.5 CPU1 CON MEMORIA SINGOLA A SINGOLO CICLO DI CLOCK 27
3.5.1 Progettazione di CPU1 con una sola memoria 27

SVILUPPO DI CPU2 32

4.1 FILOSOFIA DI PROGETTAZIONE 33
4.2 MODELLAZIONE DEI PRINCIPALI COMPONENTI 33
4.2.1 L'unità di controllo 33
4.2.2 Un registro 34
4.2.3 Un multiplexer 34
4.3 ERRORI NEGLI APPUNTI 35
4.3.1 Errori individuati 35
4.3.2 Unità di controllo corretta 36
4.3.3 Ritardo su alcuni segnali 38
4.4 TEST BENCH E SIMULAZIONE 38

UN ESEMPIO DI SIMULAZIONE 40

5.1 L'ASSEMBLATORE 41
5.2 I REGISTRI, LA MEMORIA ISTRUZIONI E DATI 42
5.3 IL PROCESSO DI SIMULAZIONE 42
5.4 VERIFICA DEL COMPORTAMENTO DEL DISPOSITIVO 44
5.5 IL PROGRAMMA 45

INDICE DELLE FIGURE E DELLE TABELLE 46

BIBLIOGRAFIA 47

Introduzione

Il VHDL è un linguaggio per descrivere i sistemi elettronici digitali sviluppato negli Stati Uniti all'interno di un progetto riguardante i circuiti integrati, iniziato nel 1980. Nasce dall'esigenza di avere uno standard per rappresentare la struttura e le funzionalità dei circuiti integrati. VHDL è un acronimo per VHSIC Hardware Description Language, dove VHSIC è a sua volta un acronimo per Very High Speed Integrated Circuits: il VHDL è pertanto un linguaggio per descrivere l'hardware, specificatamente progettato per i circuiti integrati ad alta velocità. Benché non rientri tra le sue principali caratteristiche, può essere usato per scrivere software di tipo generale ed è impiegato per la documentazione, la verifica e la sintesi dei sistemi digitali. È stato standardizzato dall'IEEE e le principali revisioni sono: *IEEE Standard 1076-1987*, *IEEE Standard 1164* e *IEEE Standard 1076-1993*; il più usato è lo Standard 1076-1987, anche se la versione più recente sta lentamente rimpiazzando quella più datata.

Le sue caratteristiche consentono un approccio alla progettazione sia di tipo *Top-Down*, sia di tipo *Bottom-Up*, permettendo così la fusione di queste due metodologie nello sviluppo di sistemi complessi. Sono supportati vari livelli di astrazione tra cui i principali sono:

- Descrizione strutturale
- Descrizione del flusso di dati
- Descrizione comportamentale

Il primo livello di astrazione permette la suddivisione di un progetto in diversi blocchi che vengono realizzati e connessi insieme per creare il progetto originale. Il secondo livello di astrazione permette di descrivere i circuiti indicando come gli ingressi e le uscite dei componenti o dei blocchi combinatori sono connessi insie-

me. In altre parole, viene descritto il modo in cui i segnali, cioè i dati, fluiscono attraverso il circuito. Il terzo livello di astrazione è differente dai due precedenti nel senso che non necessariamente riflette come è implementato il progetto: ogni componente è una scatola nera con i suoi ingressi e le sue uscite, ma è irrilevante che cosa c'è dentro. Questo può essere utile quando non è di interesse o non è disponibile la struttura interna di un modulo, per esempio quando si sta utilizzando un componente elettronico disponibile sul mercato di cui sono note solo le sue caratteristiche ma non come è stato realizzato.

Il VHDL è quindi stato progettato per soddisfare un certo numero di bisogni all'interno del processo di progettazione. Principalmente permette la descrizione della struttura di un progetto, cioè di come un progetto è decomposto in sottoprogetti e come questi sottoprogetti sono interconnessi. In secondo luogo, permette la specifica della funzionalità di un modulo usando la forma di un linguaggio di programmazione. Infine, permette di simulare un progetto prima che sia realizzato, così che il progettista può comparare velocemente le varie alternative senza i ritardi ed i costi che comporterebbe la realizzazione di un prototipo hardware.

1.1 Un esempio

Per capire meglio quali strumenti fornisce VHDL, riportiamo la descrizione di un circuito che genera un segnale di clock. Un sistema elettronico digitale può essere descritto come un modulo con ingressi ed uscite, in cui le uscite sono funzione degli ingressi. Nella terminologia VHDL, un modulo viene chiamato un'*entità*, mentre i suoi ingressi e le sue uscite sono chiamate *porte*. Le linee che connettono le porte sono chiamate *segnali*. Riportiamo la descrizione del circuito, specificando la sua interfaccia esterna che include una descrizione delle sue porte:

```
entity generatore_clock is
  generic (periodo_clock : time := 100 ns);
  port (clock : out bit);
end generatore_clock;
```

La parola chiave `entity` introduce la dichiarazione di un modulo chiamato `generatore_clock`; tale modulo ha una sola porta di uscita chiamata `clock` di tipo `bit`; il tipo `bit` garantisce che il segnale associato a quella porta possa assumere solo i due valori `'0'` e `'1'`. Viene dichiarata anche una costante che rappresenta la durata del periodo del clock. L'interfaccia esterna non specifica come opera l'entità, per cui sono necessarie le seguenti linee di codice, che descrivono l'*architettura* del circuito:

```
architecture comportamento of generatore_clock is
begin

  clock_driver : process
  begin
        clock <= '0';
        wait for periodo_clock / 2;
        clock <= '1';
        wait for periodo_clock / 2;
  end process clock_driver;

end comportamento;
```

Figura 1: Un esempio di descrizione VHDL.

La precedente descrizione è di tipo *comportamentale* in quanto non specifica come sia fatto il clock al suo interno ma bensì qual sia il suo comportamento. Il costrutto chiave per la descrizioni comportamentali fa uso della parola chiave `process`, mentre l'istruzione `architecture` è comune a tutti i tipi di descrizione. Alla uscita `clock` dell'entità `generatore_clock` viene assegnato il valore `'0'` per una durata pari alla metà del periodo di clock, poi gli viene assegnato `'1'` per l'altra metà. La sintassi del linguaggio è molto più vicina a quella del Pascal piuttosto che a quella del C.

Principali caratteristiche del linguaggio

Il VHDL è un linguaggio ampio e complesso. Poiché è stato inizialmente creato per una modellazione estremamente precisa dei circuiti per la simulazione temporale, include molte caratteristiche legate alla modellazione che sono di scarso interesse per la maggior parte degli utenti. Andremo ad illustrare solo le caratteristiche salienti del linguaggio, trascurando gli aspetti meno significativi: questa sottoparte del linguaggio ci permetterà comunque di descrivere in modo preciso tutti i tipi di sistemi.

2.1 Entità ed Architetture

Ogni descrizione in VHDL consiste almeno in una coppia entità architettura. Una dichiarazione di entità descrive il circuito come appare dall'esterno, dalla prospettiva della sua interfaccia di ingressi ed uscite. Facendo un paragone con uno schema circuitale, si potrebbe pensare ad una dichiarazione di un'entità come ad un simbolo di un blocco su di uno schema.

La seconda parte di una descrizione minimale di un progetto è la dichiarazione di un'architettura. Ogni entità deve essere completata con una corrispondente architettura. L'architettura descrive il funzionamento dell'entità a cui è riferita. Usando lo schema circuitale come metafora, una dichiarazione di una architettura è analoga ad uno schema di basso livello a cui fa riferimento un simbolo di un blocco funzionale di più alto livello.

2.1.1 Dichiarazione di un'Entità

Una dichiarazione di un'entità fornisce l'interfaccia completa di un circuito. Usando le informazioni fornite in una dichiarazione di entità, cioè i nomi, il tipo di dato e la direzione di ogni porta, si hanno a disposizione tutte le informazioni necessarie per connettere quella parte di circuito in un altro circuito di più alto livello o per testare la sua funzionalità generando opportuni segnali di ingresso. Ciò che fa realmente il circuito non è incluso in una dichiarazione di entità.

Riportiamo una semplice dichiarazione di un'entità per un circuito comparatore a 8 bit:

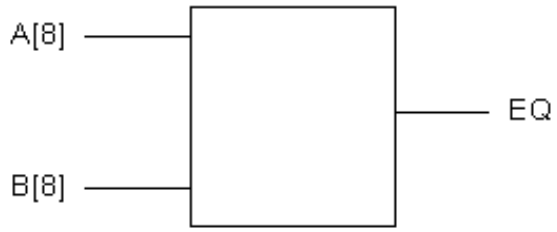


Figura 2: Un semplice comparatore.

```
entity compare is
  port (A, B : in bit_vector(0 to 7);
        EQ : out bit);
end compare;
```

Figura 3: Dichiarazione di un'entità.

La dichiarazione di un'entità comincia con la parola chiave `entity`, include un nome, `compare`, ed un'istruzione `port` che definisce tutti gli ingressi e le uscite dell'entità. L'elenco di porte comprende tre porte: `A`, `B` ed `EQ`. Per ognuna di queste porte è specificata una direzione (`in`, `out` o `inout`) ed un tipo (in questo caso `bit_vector(0 to 7)` che specifica un vettore di 8 `bit` e `bit` che rappresenta un segnale che può assumere i due soli valori `'1'` e `'0'`). Le parole chiave del linguaggio compaiono in grassetto nelle sezioni dedicate agli esempi di codice VHDL.

2.1.2 Dichiarazione e corpo di un'Architettura

La seconda parte di un file sorgente VHDL minimale è la dichiarazione di una architettura. Ogni dichiarazione di un'entità deve essere accompagnata da almeno un'architettura, ma può averne anche più di una. Ecco una dichiarazione di architettura per il circuito comparatore:

```
architecture compare1 of compare is
begin
  EQ <= '1' when (A = B) else '0';
end compare1;
```

Figura 4: Dichiarazione di un'architettura.

La dichiarazione dell'architettura comincia con un nome `compare1` che deve essere unico, seguito dal nome dell'entità alla quale è legata l'architettura, in questo caso `compare`. Nella dichiarazione dell'architettura (tra le parole chiavi `begin` ed `end`) si trova la descrizione comportamentale del comparatore. Ci sono molti modi per descrivere la logica combinatoria in VHDL: il metodo usato in questo semplice esempio fa uso di un'istruzione concorrente chiamata assegnamento condizionale. Questo assegnamento specifica che il valore dell'uscita `EQ` sarà `'1'` quando `A` e `B` sono uguali e `'0'` quando differiscono. Questo singolo assegnamento concorrente illustra la più semplice forma di architettura ma in VHDL ci molti differenti tipi di istruzioni concorrenti. Altre caratteristiche del VHDL, come i sottoprogrammi e le relazioni gerarchiche, permettono di includere componenti di più basso livello e funzioni in un'architettura e l'istruzione `process` consente di descrivere anche la logica sequenziale molto complessa.

2.2 Tipi di Dato

Come un linguaggio di programmazione di alto livello, il VHDL permette la rappresentazione dei dati attraverso tipi di dato di alto livello. Questi tipi di dato possono rappresentare delle linee singole in un circuito od un insieme di linee facendo ricorso al concetto di array.

La precedente descrizione del comparatore usa i due tipi `bit` e `bit_vector` per modellare gli ingressi e le uscite. Il tipo `bit` (`bit_vector` è semplicemente un vettore di `bit`) può assumere solo i valori `'0'` e `'1'`. Per ogni tipo di dato esiste un insieme di valori che può assumere ed un insieme di operazioni ammissibili su quei valori. Inoltre è un linguaggio fortemente tipizzato e non è possibile, ad esempio, assegnare direttamente il valore di un intero ad un vettore di `bit`: questo può essere fatto solo usando un'apposita funzione di conversione. I più comuni tipi di dato sono riportati di seguito:

- **`bit`**: un valore di un `bit` rappresenta una linea di un circuito;
- **`bit_vector`**: è un vettore di `bit`;
- **`boolean`**: può assumere solo i due valori `True` e `False`;
- **`integer`**: valore intero con segno, tipicamente implementato come tipo di dato a 32 bit;
- **`real`**: rappresenta un valore in virgola mobile;
- **`enumerated`**: viene usato per creare nuovi tipi di dato come una lista di vari elementi;
- **`record`**: raggruppa tipi di dato differenti in un unico insieme;
- **`array`**: viene usato per creare dei vettori unidimensionali e multidimensionali;
- **`access`**: simile ai puntatori in C o in Pascal;
- **`file`**: usato per leggere e scrivere file su disco: molto utile per la simulazione;
- **`physical`**: usato per rappresentare grandezze fisiche come il tempo o i volt facendo uso di unità di misura simboliche come '`nsec`' o '`ma`'.

2.3 Unità di progetto

Un concetto unico del VHDL rispetto ai linguaggi di programmazione è il concetto di *unità di progetto*. le unità di progetto sono delle parti di codice che possono essere compilate separatamente e memorizzate in una libreria. Un'entità

ed un'architettura sono due unità di progetto ma vi sono altre tre unità di progetto: pacchetti, corpo dei pacchetti e configurazioni.

2.3.1 Entità

Un'entità in VHDL definisce le specifiche esterne di un circuito. Una qualunque descrizione VHDL deve contenere almeno un'entità e la corrispondente architettura. In una dichiarazione di entità il nome deve essere unico, seguito da un elenco di porte che definiscono gli ingressi e le uscite del circuito. Per ogni porta si deve specificare un nome, una direzione ed un tipo. Opzionalmente può essere fornita una lista di parametri chiamata `generic` che permette di passare informazioni addizionali all'entità.

2.3.2 Architetture

La dichiarazione di un'architettura descrive la funzionalità o la struttura di un circuito. Ogni architettura deve essere associata con un'entità del progetto. Il VHDL permette di creare più di un'architettura per ogni entità. Questa caratteristica è particolarmente utile per la simulazione e per quei gruppi di lavoro nei quali il progetto delle interfacce è fatto da alcuni membri che sono diversi da quelli che realizzano le architetture di più basso livello. Una dichiarazione di architettura consiste in una o più dichiarazioni di segnali intermedi, di componenti, di funzioni e procedure locali e di costanti seguite da un'istruzione `begin` e da una serie di istruzioni concorrenti.

2.3.3 Pacchetti e corpo dei pacchetti

Una dichiarazione di un pacchetto è identificata dalla parola chiave `package` ed è comunemente usata per raggruppare le dichiarazioni usate da diverse unità di progetto. Un pacchetto può essere visto come un'area comune usata per memorizzare dichiarazioni di tipo, costanti e sottoprogrammi globali. Un pacchetto può essere composto di due parti basilari: una dichiarazione di pacchetto ed un corpo del pacchetto opzionale. Una dichiarazione di un pacchetto può contenere i seguenti tipi di istruzioni:

- dichiarazioni di tipi e sottotipi;
- dichiarazioni di costanti;
- dichiarazioni di segnali globali;
- dichiarazioni di procedure e funzioni;
- specificazioni di attributi;
- dichiarazioni di file;
- dichiarazioni di componenti;
- dichiarazioni di alias;
- clausole `use`.

Gli oggetti che compaiono in una dichiarazione di un pacchetto possono essere visibili alle altre unità di progetto attraverso l'uso dell'istruzione `use`. Se un pacchetto contiene delle dichiarazioni di sottoprogrammi, cioè di funzioni o di procedure, o se definisce delle costanti di cui non è dato il valore, allora è necessario un corpo del pacchetto.

Il corpo del pacchetto, che è identificato dalla combinazione delle due parole chiavi `package body`, deve avere lo stesso nome del corrispondente pacchetto ma può essere locato dovunque nel progetto. La relazione tra pacchetto e corpo del pacchetto è simile a quella tra entità e la corrispondente architettura, con l'eccezione che ci può essere solo un corpo per ogni pacchetto. Mentre la dichiarazione del pacchetto fornisce le informazioni necessarie per usare gli oggetti in esso definiti, il comportamento di tali oggetti come le procedure e le funzioni deve essere specificato nel corpo del pacchetto.

2.3.4 Configurazioni

L'ultimo tipo di unità di progetto disponibile in VHDL è chiamata configurazione. Una dichiarazione di una configurazione può essere vista come una lista di parti del progetto; è identificata con la parola chiave `configuration` e specifica quali architetture devono essere collegate alle entità. Le dichiarazioni di configurazione sono sempre opzionali; nel caso non ci sia una dichiarazione di configurazione, lo standard specifica un insieme di regole che forniscono una configurazione di default. Per esempio, se si è fornita più di un'architettura per un'entità, l'ultima architettura compilata avrà la precedenza e si legherà all'entità.

2.4 Livelli di astrazione

Il VHDL supporta diversi stili per descrivere un progetto che differiscono principalmente per quanto sono legati allo hardware sottostante, dando vita così a vari livelli di astrazione. Per fare un esempio, è possibile descrivere un contatore in vari modi: al più basso livello di astrazione, quello *strutturale*, si potrebbe connettere una serie di porte logiche e di flip flop per formare il circuito. Il più alto livello di astrazione, chiamato *comportamentale*, descrive il comportamento del circuito al trascorrere del tempo. Il concetto del tempo delimita la descrizione comportamentale da quella di basso livello. In una descrizione comportamentale, il concetto del tempo può essere espresso precisamente, specificando i ritardi tra gli eventi, come ad esempio i ritardi di propagazione nelle porte e nelle linee, oppure può semplicemente ordinare le operazioni che sono espresse sequenzialmente come in una descrizione funzionale di un flip flop. Come nei linguaggi di programmazione guidati dagli eventi, vengono scritti uno o più programmi che operano sequenzialmente e comunicano l'un l'altro attraverso le loro interfacce.

Un metodo alternativo di progettazione suddivide il circuito in logica combinatoria e registri ed è chiamato livello di astrazione del *flusso di dati*. Tale livello di astrazione è di livello intermedio e permette di nascondere la logica combinatoria mentre le parti più importanti del circuito, come ad esempio i registri, sono completamente specificate. Per collocarsi a questo livello di astrazione, è necessario innanzitutto dare una descrizione dei registri che si intendono usare; poi questi elementi devono essere forniti come componenti e inseriti nella gerarchia che forma il progetto oppure in forma di sottoprogrammi come funzioni o procedure.

I livelli di astrazione comportamentale e del flusso di dati descrivono il circuito in termini della sua funzione logica. C'è un terzo livello che è usato per combinare insieme varie descrizioni in una descrizione gerarchica del circuito. Il livello di astrazione strutturale permette di incapsulare una parte di un progetto come un semplice flip flop o un più grande sottocircuito come un componente riusabile ed è analogo ad uno schema testuale o ad un diagramma a blocchi per progetti di più alto livello. Una descrizione strutturale è l'equivalente di una lista di componenti e delle loro connessioni, in cui i vari componenti sono connessi in-

sieme per creare un sistema più complesso. La gerarchia di un progetto viene creata collegando insieme i vari componenti.

2.5 L'istruzione `process`

L'istruzione `process` è il mezzo primario per descrivere le operazioni sequenziali. La forma più comune di un'istruzione `process` è la seguente:

```
architecture arch_name of ent_name is
begin
  process_name : process (sensitivity_list)
        local_declaration;
        local_declaration;
        ...
  begin
        sequential statement;
        sequential statement;
        sequential statement;
        .
        .
        .
  end process;
end arch_name;
```

Figura 5: L'istruzione `process`.

Un'istruzione `process` è costituita dalle seguenti parti:

- Un nome di processo opzionle, cioè un identificatore seguito dai due punti.
- La parola chiave `process`.
- Una lista di sensitività opzionale, che indica quali segnali portano alla esecuzione del processo quando avviene un certo evento. La lista di sensitività è richiesta se il processo non include nessuna istruzione `wait` che ne sospende l'esecuzione.
- Una sezione di dichiarazione opzionale che permette di definire oggetti e sottoprogrammi locali.
- La parola chiave `begin`.

- Una sequenza di istruzioni che devono essere eseguite quando il programma è in esecuzione.
- Un'istruzione `end process` che termina il processo.

Un processo può essere paragonato al software guidato dagli eventi: così come un programma viene eseguito quando avviene un evento sui suoi ingressi, il processo viene attivato per la simulazione quando avviene un evento sui segnali della lista di sensitività. Un processo descrive l'esecuzione sequenziale di istruzioni che sono dipendenti da uno o più eventi che sono avvenuti. Un flip flop rappresenta bene questa situazione: rimane inattivo, senza cambiare stato, finché non avviene un evento significativo che lo fa attivare e cambiare di stato. Benché ci sia un ordine predefinito nell'eseguire le operazioni di un processo, cioè dallo inizio alla fine, si deve pensare che un processo esegua in tempo zero. Questo significa che un processo può essere usato per descrivere funzionalmente un circuito senza tener conto della reale temporizzazione e processi multipli possono essere eseguiti in parallelo senza tener conto di quale terminerà prima le proprie operazioni. Riportiamo adesso un esempio di processo:

```
reg : process(Rst, Clk)
  variable Qreg : bit_vector(0 to 7);
begin
  if Rst = '1' then -- Reset Asincrono
        Qreg := '00000000';
  elsif (Clk = '1' and Clk'event) then
        if (Load = '1') then
              Qreg := Data;
        else
              Qreg := Qreg(1 to 7) & Qreg(0);
        end if;
  end if;
  Q <= Qreg
end process;
```

Figura 6: Un esempio di processo.

Il processo è dipendenti dai due input asincroni `Clk` e `Rst`. Questi segnali sono i soli segnali i cui eventi possono direttamente aver effetto sulle operazioni del circuito; in assenza di un evento su questi segnali, il circuito descritto dal processo

manterrà semplicemente il suo valore corrente, cioè rimarrà sospeso. Esaminiamo adesso che cosa accade quando avviene un evento su uno dei due input asincroni. Per prima cosa consideriamo il caso in cui avviene una transizione ad `'1'` sul segnale `Rst`. In questo caso, il processo comincerà l'esecuzione e sarà valutata la prima istruzione `if`. Poiché è avvenuta una transizione ad `'1'`, il simulatore vedrà che la condizione `Rst = '1'` è vera e assegnerà alla variabile `Qreg` il valore di reset `"00000000"`. Le rimanenti istruzioni dell'espressioni `if-then-elsif` saranno ignorate. L'istruzione finale, l'assegnamento al segnale di usita `Q` del valore di `Qreg`, non è soggetta all'espressione `if-then-elsif` ed è piazzata nella coda del processo per l'esecuzione (gli assegnamenti tra segnali non avvengono finché il processo non si ferma). Infine il processo si ferma e tutti i segnali a cui erano stati assegnati dei valori sono aggiornati e si attende un nuovo evento. Supponiamo adesso che avvenga un evento su `Clk`. In questo caso, il processo eseguirà e sarà valutata l'espressione `if-then-elsif` finché non si trova una condizione valida. Se il segnale `Rst` continua ad avere un valore alto, allora il simulatore valuterà vero il primo `if` e la condizione di reset sarà eseguita. Se `Rst` non ha un valore alto, sarà valutata l'espressione `Clk = '1' and Clk'event`. Questa espressione è comunemente usata per individuare i cambiamenti del clock da basso ad alto. Infatti la sola istruzione `Clk = '1'` non è abbastanza specifica, poiché il processo può eseguire a causa di un evento su `Rst` che non risulta in un cambiamento a `'1'` di `Rst`. Per assicurare che l'evento a cui stiamo rispondendo è di fatto un evento sul clock, si fa uso dell'attributo `event` che verifica se è avvenuto un cambiamento su quel segnale. Se l'evento che ha richiamato il processo è un fronte di salita di `Clk`, allora il simulatore testerà la restante istruzione `if-then` per determinare quale istruzione di assegnamento deve essere eseguita. Se `Load` vale `'1'`, viene eseguita la prima istruzione di assegnamento ed il dato viene caricato su `Qreg`; altrimenti il dato viene shiftato facendo uso dello operatore di concatenazione `&`.

2.5.1 L'istruzione `process` senza lista di sensitività

Ci sono due forme di processo in VHDL. La prima forma usa la lista di sensitività ed esegue durante la simulazione ogni volta che avviene un evento su un segnale

appartenente a tale lista. Questa è la forma più comune per descrivere la logica sequenziale. C'è comunque un'altra forma dell'istruzione, utile in altre applicazioni, che non include la lista di sensitività ma che include una o più istruzioni che sospendono l'esecuzione del processo fino a che non si raggiunge una determinata condizione. Il miglior esempio di un tale processo è un *test bench*, nel quale una sequenza di segnali di ingresso è applicata al trascorre del tempo ed è definito un preciso valore temporale per la riattivazione del processo. La forma generale di tale processo è la seguente:

```
architecture arch_name of ent_name is
begin
  process_name : process
        local_declaration;
        local_declaration;
        ...
  begin
        sequential statement;
        sequential statement;
        sequential statement;
        ...
        wait for (time);
        ...
  end process;
end arch_name;
```

Figura 7: L'istruzione `process` senza lista di sensitività.

Il VHDL richiede che tutti i processi abbiano una lista di sensitività oppure che includano una o più istruzioni `wait` che sospendano il processo. Non è legale avere contemporaneamente una lista di sensitività e un'istruzione `wait`.

2.6 VHDL Concorrente e Sequenziale

La differenza fondamentale tra istruzioni concorrenti e sequenziali è il concetto più importante che rende efficace l'uso del linguaggio. Il seguente diagramma illustra la differenza fondamentale tra questi due tipi di istruzioni:

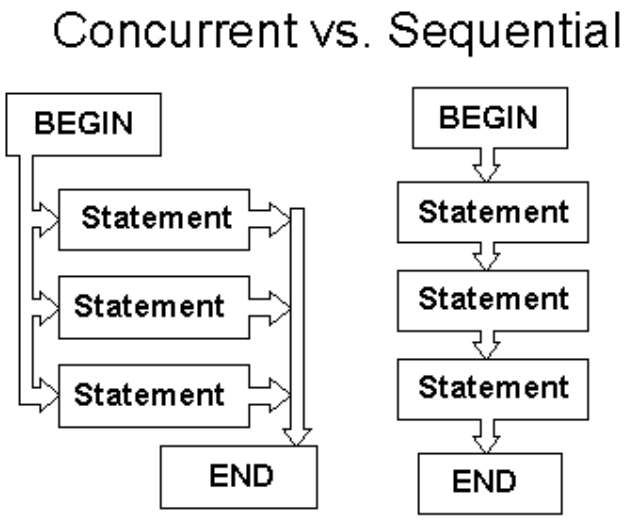


Figura 8: Istruzioni concorrenti e sequenziali.

La parte sinistra del diagramma illustra come vengono eseguite le istruzioni concorrenti. Le istruzioni concorrenti sono quelle istruzioni che compaiono tra le istruzioni `begin` ed `end` di una architettura. Questa parte dell'architettura è conosciuta come l'*area concorrente*. In VHDL, tutte le istruzioni nell'area concorrente sono eseguite allo stesso istante e non è significativo l'ordine nel quale compaiono le istruzioni. Le istruzioni sequenziali sono invece quelle che compaiono allo interno di un processo, in una funzione o in una procedura: la loro caratteristica è che vengono eseguite nell'ordine in cui compaiono. Un processo è un'istruzione concorrente singola in quanto viene eseguita in tempo zero: le sue istruzioni sono però eseguite dalla prima all'ultima.

2.7 Test Bench

I test bench completano la descrizione di un progetto: infatti non è sufficiente descrivere il comportamento del circuito dall'interno all'esterno perché il solo modo per verificare che una descrizione di un progetto scritta in VHDL operi come atteso, è di simularla. I test bench forniscono l'ambiente in cui eseguire la simulazione. Il modo più semplice di pensare ad un test bench è di immaginarlo come un circuito che applica stimoli alla descrizione del progetto e che opzionalmente verifica anche se il circuito simulato si comporta nel modo atteso. La se-

guente figura illustra la relazione tra un test bench e la descrizione di un progetto, che viene chiamata *unità* o *dispositivo sotto test*.

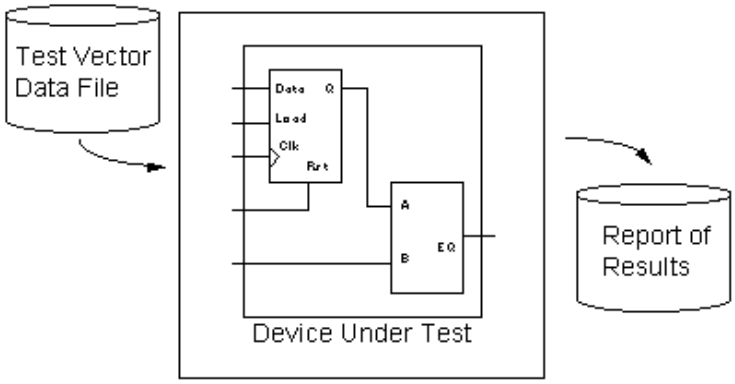


Figura 9: Un esempio di test bench.

Per applicare degli stimoli al progetto, il test bench sarà probabilmente scritto usando uno o più processi in sequenza, ed userà una serie di assegnamenti tra segnali e di istruzioni `wait` per descrivere gli stimoli. Si possono usare istruzioni di loop per descrivere gli stimoli ripetitivi ed usare i file ed i record per applicare gli stimoli nella forma di *vettori di test*. Per testare il risultato della simulazione, si può usare l'istruzione `assert` oppure scrivere i risultati su file per un'analisi successiva. Per le descrizioni di progetti complessi, lo sviluppo di un test bench può diventare esso stesso un progetto di ampia scala e spesso i test bench sono più grandi e complessi del progetto stesso. Per questo si dovrebbe pianificare il progetto in modo da tener conto del tempo richiesto per sviluppare il test funzionale. Una buona idea è quella di creare test bench riutilizzabili, che leggono l'input da file. Un approccio comune per creare degli stimoli comprensibili è quello di creare un file con gli ingressi insieme ai valori attesi per quegli ingressi. Queste tabelle di vettori sono chiamate vettori di test e possono essere usati ripetitivamente per verificare che un circuito complesso continui a funzionare dopo che sono state apportate alcune modifiche.

Sviluppo di CPU1

La CPU1 è un prototipo che si ispira alle macchine RISC costruito in modo da eseguire le istruzioni in un solo periodo di clock. L'esecuzione di tutte le istruzioni in un solo periodo di clock comporta delle problematiche riguardo alla propagazione dei segnali, soprattutto di quelli che comandano la memoria. Per questo sono stati realizzati più modelli di questa CPU, ognuno con determinate caratteristiche. Il primo modello, riportato negli Appunti, non opera correttamente a causa dei ritardi di propagazione che non originano dei segnali corretti per controllare la memoria dati. Il secondo modello risolve il precedente inconveniente facendo uso del segnale di clock per guidare i segnali diretti alla memoria dati. Il terzo modello accorpa la memoria dati e la memoria istruzioni in una sola memoria, giocando sui due semiperiodi di clock, introducendo il registro IR e ritardando in modo opportuno i segnali che controllano la memoria.

3.1 Filosofia di progettazione

La realizzazione di CPU1 segue un approccio di progettazione di tipo Bottom-Up: prima sono stati realizzati i vari componenti, poi questi componenti sono stati assemblati in unità logiche che collegate insieme modellano completamente la CPU. La progettazione della CPU è stata quindi suddivisa in molti sottoprogetti di unità più semplici facili da realizzare. La modellazione di queste unità più semplici segue un approccio comportamentale, mentre quella delle unità logiche le raggruppano e quella della CPU seguono un approccio di tipo strutturale. In questo modo si ottiene così una CPU organizzata a vari livelli di dettaglio.

Le tre unità logiche che compongono CPU1 sono la sezione di fetch, la sezione di decodifica e la sezione di esecuzione dell'istruzione. L'interconnessione di queste unità genera la CPU. Queste tre unità sono composte da vari componenti, come il register file (RF), l'ALU, il program counter (PC), i multiplexer ed altri ancora. Oltre ai componenti interni alla CPU, abbiamo un modulo che genera il segnale di clock e, dipendentemente dalla realizzazione, una memoria istruzioni e una memoria dati o solo una memoria che contiene istruzioni e dati.

3.2 Modellazione dei principali componenti

I componenti base che concorrono a formare la CPU sono modellati in modo comportamentale, cioè non viene descritto come sono fatti all'interno ma solo come si comportano da un punto di vista esterno. Dopo aver dichiarato la entità che specifica quali sono gli ingressi e le uscite, la corrispondente architettura è costituita da un solo processo che descrive il comportamento del modulo.

La struttura di un test bench si ripete per i vari componenti e può essere schematizzata nel seguente modo. Si dichiara un'entità che rappresenta il test bench: il test bench non ha né ingressi, né uscite. S definisce poi l'architettura associata all'entità: questa architettura conterrà sempre come componente l'entità che si vuole testare e può essere seguita da altri componenti che servono ad inserire il dispositivo nell'ambiente in cui deve operare. Il test bench prosegue con la dichiarazione dei segnali utili a collegare il componente sotto test. Nella sezione successiva si collegano i vari moduli con i segnali dichiarati. L'architettura viene completata con un processo che descrive il modo in cui devono variare gli ingressi: tale processo contiene delle istruzioni `wait` che temporizzato l'applicazione dei vari segnali di test. Il test bench si conclude con la dichiarazione di una configurazione che specifica quali entità devono essere usate per modellare i componenti dichiarati all'interno dell'architettura.

3.2.1 I nomi dei file

Ogni componente è descritto in un file separato che ha lo stesso nome del componente, questo per ritrovarlo con facilità e per evitare di dover compilare tutto ogni

volta: ad esempio, se il componente si chiama `ALU`, allora il file che lo contiene si chiamerà `ALU.vhd`. Per poter simulare un componente, occorre scrivere un test bench: il nome file contenente il test bench avrà come prefisso il nome del componente e sarà seguito dal suffisso `_test`. Continuando con il precedente esempio, il file contenete il test bench per l'`ALU` si chiamerà `ALU_test.vhd`. Inoltre, per poter simulare un componente, occorre un file che comunichi al simulatore quali sono i segnali da visualizzare e per quanto tempo eseguire la simulazione: tale file si chiamerà `ALU.cmd`. I comandi contenuti in questo file possono anche essere scritti uno ad uno all'interno del simulatore, ma tale pratica non risulta possibile se i segnali da disegnare sono in numero considerevole.

3.2.2 Il pacchetto `cpu_tipi`

Questo pacchetto contiene le definizione di nuovi tipi, costanti e funzioni di utilità generale. Sono definite due tabelle che permettono di convertire valori di tipo `bit` in valori di tipo `boolean` e viceversa. Per standardizzare i nomi dei vettori di `bit`, con `bit_n` si indica un vettore di `bit` n-dimensionale. Nel pacchetto si dichiarano anche le costanti che rappresentano il codice di una generica operazione ed il codice di operazione aritmetica. Le funzioni si occupano di convertire valori interi in vettori di bit, di eseguire le operazioni aritmetiche e logiche. La istruzione:

```
use work.cpu_tipi.all;
```

permette di importare il pacchetto in una qualunque unità di progetto.

3.2.3 Il register file

Il register file (RF) è un banco di registri costituito da 32 registri di 32 bit ciascuno di uso generale. Il processo che modella il comportamento del RF viene attivato ogni volta che avviene un evento sui segnali `CLOCK`, `RR1` e `RR2`. Alla prima esecuzione del processo, si legge da file il contenuto dei 32 registri: il registro zero deve sempre contenere il valore zero. Durante le esecuzioni successive, per prima cosa si verifica se è avvenuto un evento che ha portato a zero il `CLOCK`: se

siamo in questo caso, si esamina se è asserito il comando di scrittura `RWrite` e se lo è, si scrive il dato `D` nel registro destinazione individuato da `RW` e si aggiornano le due uscite `A` e `B` con il contenuto dei registri `RR1` e `RR2`. Nel caso non si sia verificato nessun evento cha abbia portato a zero il `CLOCK`, si aggiornano le due uscite a seconda di quale segnale è cambiato tra `RR1` e `RR2` Per concludere il processo che modella il RF, si scrive su file un'immagine di tutti i registri che sarà molto utile in fase di simulazione per verificare che tutto si sia svolto in modo corretto. Tutte le operazioni vengono eseguite tenendo conto del ritardo di questo dispositivo: negli Appunti si parla di 5 nsec per la decodifica dell'istruzione e per la lettura dei registri. Nel modello realizzato, i 5nsec di ritardo sono stati attribuiti tutti al RF.

3.2.4 La ALU

Il processo che descrive il comportamento della ALU è semplice: si attiva ogni volta che è avvenuto un evento su uno dei due operandi `A` e `B` oppure su uno dei segnali che indicano quale operazione deve essere effettuata. Una volta attivato, si verifica in modo sequenziale quale segnale di operazione è asserito: si esegue poi l'operazione richiesta tramite una chiamata a funzione e si settano i bit `Zero`, `Segno` e `Overflow`: tutto questo tenendo conto del ritardo interno della ALU. Nella CPU1 con una sola memoria che esegue le istruzioni in un ciclo di clock è necessario un'ulteriore uscita che specifica se è avvenuto un overflow nel calcolo dell'indirizzo. Nella CPU2 questa uscita non è più necessaria in quanto il tipo di overflow può essere individuato dallo stato in cui avviene. Il segnale `Segno` serve ad aggiungere l'istruzione "salta se minore", utile per poter scrivere un programma che faccia qualcosa di sensato. L'espressione logica del segnale che pilota il multiplexer che stabilisce come deve essere modificato il PC, si modifica nel seguente modo: (`Zero` AND `JE`) OR (`Segno` AND `JS`), dove `JS` è l'uscita dal decodificatore del codice operativo dell'istruzione relativa all'operazione "salta se minore". Inoltre i segnali `JE` e `JS` sono messi in OR per fornire al modulo `ContrALU` il segnale appropriato, così come viene fatto per `LD` e `ST`.

3.2.5 La memoria

La memoria rappresenta il componente di più difficile modellazione. Nelle CPU in cui sono presenti la memoria istruzioni e la memoria dati separate, la memoria istruzioni viene modellata in modo molto semplice con un processo che viene attivato ogni volta che cambia l'indirizzo dell'istruzione. Negli altri casi, cioè quando la memoria è unica o quando si modella la memoria dati, risulta molto complesso capire quali sono i segnali che devono attivare il processo che la descrive. Con la memoria descritta negli Appunti, non si riesce a costruire una CPU che esegua tutte le istruzioni in un solo clock se non ritardando in modo opportuno i segnali di scrittura e di lettura. Inoltre è difficoltoso capire quando si deve leggere o scrivere un dato: ovviamente quando i segnali `Mread` e `Mwrite` passano da zero ad uno, allora deve essere letto o scritto un dato; ma, ad esempio, se mentre il segnale di scrittura è asserito avviene un evento sull'indirizzo, il dato deve essere scritto nella nuova locazione di memoria oppure no? Oppure se cambia il dato e l'indirizzo rimane lo stesso? Nel primo modello costruito, i segnali facenti parte della lista di sensitività del processo della memoria sono `Mread`, `Mwrite`, `Indirizzo` e `DatiIn`. Questo porta ad un prototipo di CPU1 che non opera correttamente, come sarà meglio descritto nel seguente paragrafo. Una volta stabiliti i segnali che attivano il processo, la modellazione della memoria risulta semplice. La prima volta che viene eseguito il processo, si caricano da file le istruzioni e i dati nelle opportune locazioni. Durante le successive attivazioni, se `Mread` è asserito si legge il dato e lo si presenta in uscita, altrimenti se è asserito `Mwrite` si scrive il dato in ingresso all'indirizzo opportuno. Se nessuno dei due segnali è asserito, non avviene nessun accesso alla memoria ed il dato in uscita vale `"0000_0000"`: tutto questo tenendo conto del ritardo di propagazione di 30 nsec. Il processo termina scrivendo su file un'immagine della memoria. Gli altri modelli di memoria realizzati, danno vita alle versioni funzionati della CPU1 e saranno descritti in modo approfondito nel paragrafo di competenza.

3.3 CPU1 descritta negli Appunti

Dopo aver descritto i principali componenti che servono per costruire una generica CPU, si passa allo loro connessione. L'assemblamento di tali componenti origina tre sottoparti in cui può essere suddivisa logicamente la CPU1: la sezione di fetch, la sezione di decodifica e la sezione di esecuzione di un'istruzione. Queste tre sezioni sono descritte in modo strutturale, elencando all'interno della loro architettura quali sono i componenti che occorrono e in che modo sono connessi attraverso i vari segnali. Componendo queste tre sezioni, si giunge ad una descrizione strutturale della CPU1. Il test bench fa uso di una memoria istruzioni, di una memoria dati e di un modulo che genera il segnale di clock. Questi elementi vengono collegati in modo opportuno, le due memorie e il RF sono inizializzati con i valori letti dai rispettivi file.

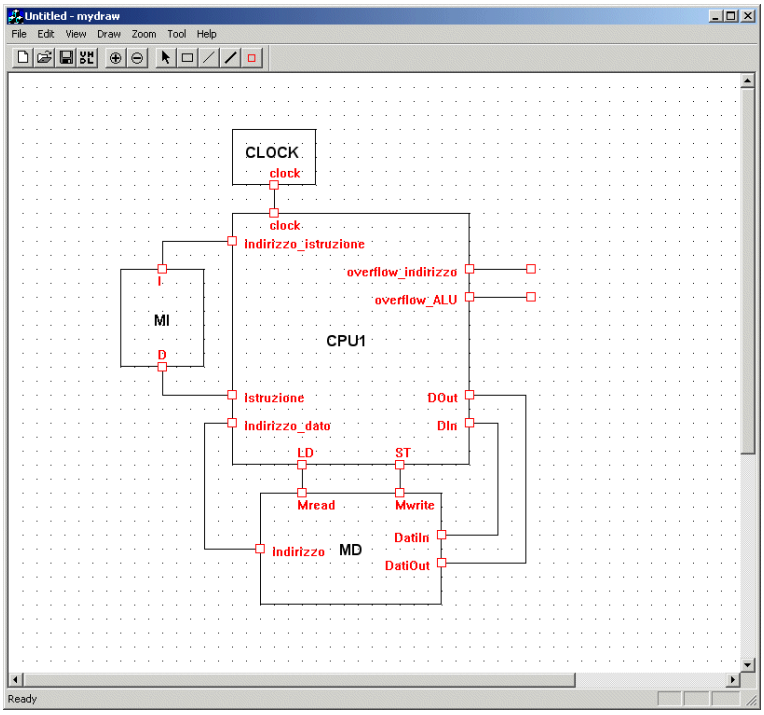


Figura 10: Test bench per la CPU1 come descritta negli appunti.

Costruito il test bench, non resta che simulare la CPU1 e vedere se il modulo si comporta nel modo atteso.

3.3.1 L'errore

Si riporta di seguito un'immagine di una simulazione che evidenzia un problema nella propagazione dei segnali di lettura e di scrittura della memoria dati.

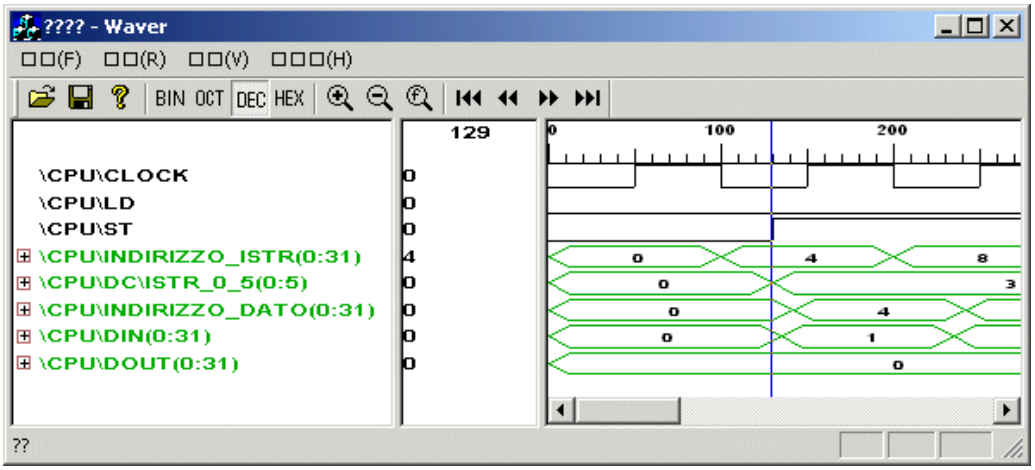


Figura 11: Errore della CPU1: la linea blu indica l'istante in cui avviene la scrittura.

Il processo di esecuzione di un'istruzione inizia con la lettura dell'istruzione dalla memoria. Questo primo passo introduce un ritardo di 30 nsec. Poi l'istruzione viene decodificata. Se si tratta ad esempio di un'istruzione di store che memorizza il contenuto di un registro nella memoria dati, alcuni campi dell'istruzione vanno in ingresso al RF e successivamente alla ALU che calcola l'indirizzo in cui deve essere memorizzato il contenuto del registro: prima che l'indirizzo sia valido, passano altri 17 nsec dovuti ai tempi di propagazione nel RF e nell'ALU. D'altra parte, il codice operativo dell'istruzione va in ingresso ad un decodificatore che con un ritardo molto inferiore ai 17 nsec asserisce il comando `Mwrite` della memoria dati. Quindi viene scritto nella memoria un dato che non è valido, in più ad un indirizzo certamente sbagliato. Ulteriori problemi nascono dalla possibilità di eseguire due istruzioni di store in sequenza: il segnale di scrittura della memoria rimane sempre asserito mentre invece l'indirizzo potrebbe cambiare: andrebbe quindi prevista la possibilità di scrivere nella memoria anche quando varia solo l'indirizzo nel caso il comando di scrittura rimanga sempre asserito. Il modello della memoria diventa logicamente complesso e non di facile comprensione.

3.4 CPU1 corretta

Una modifica che può essere apportata per rendere funzionante questo prototipo che esegue le istruzioni in un solo periodo di clock, consiste nell'usare il clock per pilotare i segnali che controllano la memoria dati. Il clock è suddiviso in due semiperiodi: il primo semiperiodo è basso mentre il secondo è alto: tutte le operazioni di aggiornamento avvengono sul fronte di discesa del clock. L'idea consiste nell'usare il fronte di salita intermedio del ciclo per eseguire alcune operazioni: in questo modo non tutte le operazioni di aggiornamento sono legate al fronte di discesa del clock. Il problema della precedente realizzazione è dovuto al fatto che i segnali di scrittura e di lettura della memoria vengono asseriti quando ancora l'indirizzo e il dato non sono validi: bisogna fare in modo che quando si accede alla memoria, l'indirizzo e il dato siano validi. Una soluzione consiste nel comandare tali segnali con il clock della CPU: mettendo i segnali di scrittura e di lettura della memoria in AND con il clock, i segnali risultanti si asseriscono al momento opportuno, cioè quando il dato e l'indirizzo sono validi, se i due semiperiodi del clock sono di durata adeguata. In particolare, il fronte di salita che separa i due semiperiodi deve verificarsi quando il dato e l'indirizzo sono oramai stabili: facendo un po' di conti, l'indirizzo e il dato sono validi solo dopo che sono trascorsi almeno 47 nsec in quando 30 nsec servono alla memoria dati per fornire in uscita l'istruzione, 5 nsec al RF per il registro base e 12 nsec per l'ALU che deve sommare il contenuto del registro base con l'offset. La durata del primo semiperiodo del clock deve essere quindi di almeno 47 nsec.

Una considerazione a parte va fatta per quanto riguarda la scrittura in un registro di un dato presente in memoria. Il segnale `Mread`, ottenuto come AND tra il segnale `LD` e il clock, si abbassa alla fine del ciclo in modo solidale con il clock. In questo stesso istante, l'uscita della memoria diventa instabile. Contemporaneamente il dato in uscita dalla memoria viene scritto nel RF, in quanto la scrittura nel RF avviene sul fronte di discesa del clock. Di conseguenza si verifica la scrittura di un dato instabile nel RF. Questo problema viene risolto ritardando il segnale `Mread`: un ritardo di 5 nsec garantisce il funzionamento della CPU1. La scrittura di un dato in memoria non genera invece alcun problema.

Per realizzare le modifiche descritte, vengono aggiunte due porte AND. Un ingresso di entrambe le porte è costituito dal clock. Per quanto riguarda la ge-

nerazione del segnale `Mwrite`, l'altro ingresso è il segnale `ST` uscente direttamente dal decodificatore dell'istruzione e l'uscita della porta AND così collegata va direttamente in ingresso alla memoria. Per quanto riguarda in vece il segnale `Mread`, il secondo ingresso è il segnale `LD` uscente direttamente dal decodificatore e l'uscita della porta AND così collegata va in ingresso ad un ritardatore di 5 nsec. Gli unici segnali che attivano il processo che descrive la memoria dati sono quindi `Mread` e `Mwrite`. La seguente figura illustra le modifiche precedentemente descritte.

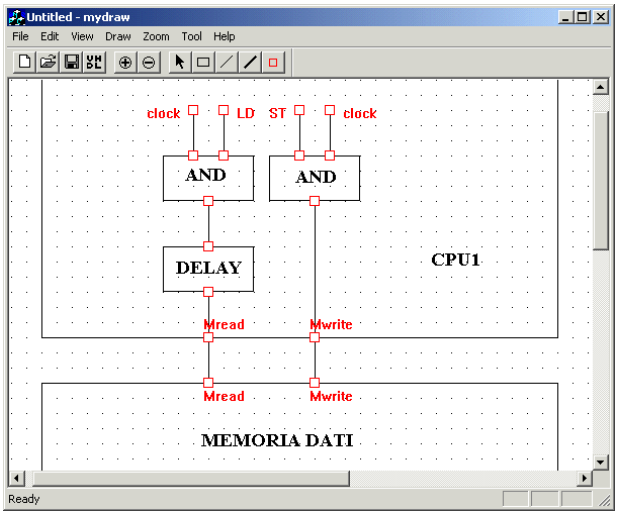


Figura 12: Generazione dei segnali `Mread` e `Mwrite`.

La Figura 13 illustra un esempio di simulazione: la linea blu evidenzia il ritardo introdotto sul segnale `Mread`.

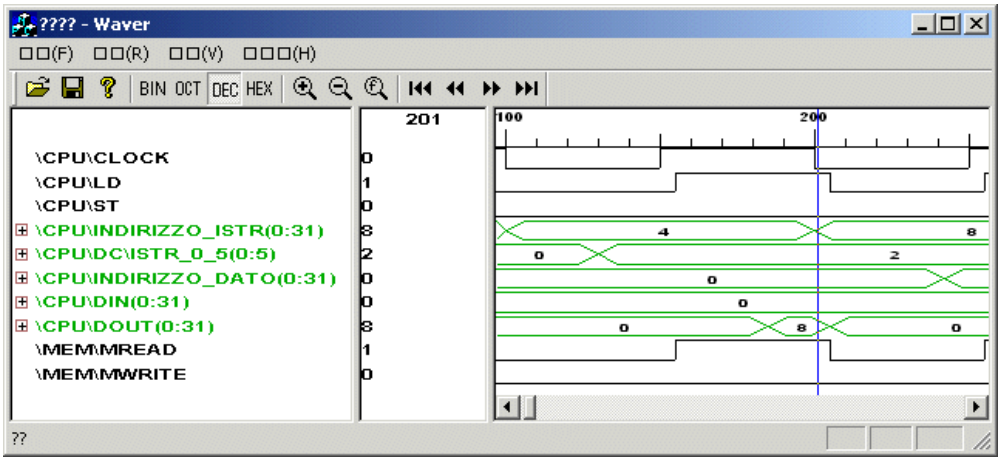


Figura 13 CPU1 con le due porte AND.

3.5 CPU1 con memoria singola a singolo ciclo di clock

Un altro prototipo di CPU1 che esegue le istruzioni in un solo ciclo di clock può essere costruito giocando sui due semiperiodi del clock senza utilizzare due memorie distinte per istruzioni e dati ed utilizzando la ALU anche per calcolare gli indirizzi delle istruzioni: sarà tuttavia necessario introdurre alcuni registri per tenere in memoria dati che non sono più stabili sul bus in uscita dalla memoria o su quello in uscita dalla ALU. Inoltre i segnali di lettura e scrittura in memoria devono essere opportunamente ritardati per attendere che il dato e l'indirizzo siano validi. Tale modello non è riportato negli Appunti e sarà quindi descritto nel successivo paragrafo.

3.5.1 Progettazione di CPU1 con una sola memoria

Si riportano di seguito le principali modifiche rispetto al modello descritto negli Appunti. La CPU1 è collegata alla memoria tramite un bus dati/istruzioni ed un bus indirizzi. Il sommatore che nella precedente realizzazione incrementava il PC non è più necessario in quanto tale operazione può essere svolta dalla ALU. Il PC viene aggiornato sul fronte di discesa del clock mentre l'IR viene aggiornato sempre sul fronte di discesa del clock ma sfasato di 50 nsec. L'uscita della ALU finisce in un registro chiamato DEST che viene aggiornato sul fronte di salita del clock: questo registro serve a memorizzare temporaneamente l'uscita della ALU per un'eventuale scrittura nel RF, in quanto l'uscita della ALU cambia quando si calcola il nuovo indirizzo. Un flip flop chiamato J pilota il multiplexer che seleziona l'incremento per il PC: l'incremento vale quattro nelle istruzioni "non di salto" e SX4 (scostamento esteso con segno) nelle istruzioni di "salto se uguale" e "salto con segno".Tale flip flop viene aggiornato sul fronte di salita del clock e serve per memorizzare durante il secondo semiperiodo del clock il valore presente durante il primo semiperiodo: i segnali `Zero` e `Segno` con i quali è pilotato cambiano quando si raggiunge il secondo semiperiodo. Sono stati aggiunti due multiplexer agli ingressi della ALU che hanno come segnale di selezione il clock: nel primo semiperiodo lasciano passare gli stessi segnali che transitavano nelle precedenti realizzazioni, mentre nel secondo periodo selezionano rispettivamente il PC e lo scostamento che gli deve essere sommato. L'uscita della ALU finisce nel re-

gistro DEST ed all'ingresso di un multiplexer pilotato dal segnale di `jmp` che serve ad aggiornare il contenuto del PC nel caso di una istruzione di salto incondizionato. Un altro multiplexer, guidato dal segnale `LD` che esce dal decodificatore della istruzione, è posto all'ingresso del dato al RF: se il segnale `LD` è basso manda in ingresso al RF l'uscita del registro DEST, mentre se il segnale `LD` è alto manda in ingresso al RF l'uscita della memoria. Il componente `ContrALU` ha in ingresso il clock che temporizza e seleziona se si deve eseguire un'operazione di somma per gli indirizzi oppure una normale operazione di ALU: quello che le differenzia una operazione di somma tra indirizzi e tra valori numerici ordinari è la condizione di overflow.

I segnali `Mread` e `Mwrite` per la memoria non sono quelli uscenti dal decodificatore dell'istruzione ma vengono costruiti nel seguente modo:

`Mwrite` = `ST` AND `clock` ritardato poi di 5 nsec;

`Mread` = `Mread_fetch` OR `Mread_load` dove

`Mread_fetch` = NOT (`clock`) AND NOT (`clock_rit`) dove

`clock_rit` è `clock` ritardato di 10 nsec;

`Mread_load` = `LD` AND `clock` ritardato di 5 nsec.

Riportiamo adesso una descrizione di come si svolgono le varie operazioni considerando un periodo di clock di 200 nsec suddiviso in ugual modo tra i due semiperiodi. Si è preso un clock di 200 nsec per avere una maggiore leggibilità nella simulazione; comunque non si può scendere sotto i 140 nsec. I valori numerici riportati si riferiscono al tempo del ciclo di clock:

0- : sul bus indirizzi è presente l'indirizzo corretto per la prossima istruzione;

0 : il clock si abbassa, `Mread` si abbassa

10 : `Mread` diventa asserito, si attiva il processo della memoria e si avvia il ciclo di lettura;

40 : arriva l'istruzione all'ingresso di IR;

50 : l'istruzione viene memorizzata in IR e contemporaneamente viene inviata al decodificatore; eventualmente si alzano i segnali `Load` o `Store`;

56 : gli ingressi alla ALU sono corretti;

68 : l'ALU termina il calcolo e produce un'uscita; si alzano eventuali segnali di `Zero`/`Segno`;

100 : si alza il clock; si memorizza la condizione di salto condizionato sul flip-flop J; si memorizza la uscita della ALU sul registro DEST; i due multiplexer in ingresso alla ALU pilotati dal clock presentano gli ingressi alla ALU provenienti dalla sezione di decodifica ed il controller della ALU invia il segnale di somma per gli indirizzi, ignorando il segnale `fALU`;

105 : si alzano l'eventuali segnali `Mread/Mwrite`

112 : l'ALU produce il nuovo indirizzo dell'istruzione, qualora non l'istruzione non sia un salto incondizionato;

135 : la memoria finisce l'eventuale ciclo di lettura/scrittura;

200 : il RF viene aggiornato con l'uscita del multiplexer pilotato dal segnale di load se il segnale `RWrite` è asserito, cioè con il dato proveniente dalla memoria oppure dal registro DEST.

Di seguito si riporta il test bench della CPU1 con una sola memoria:

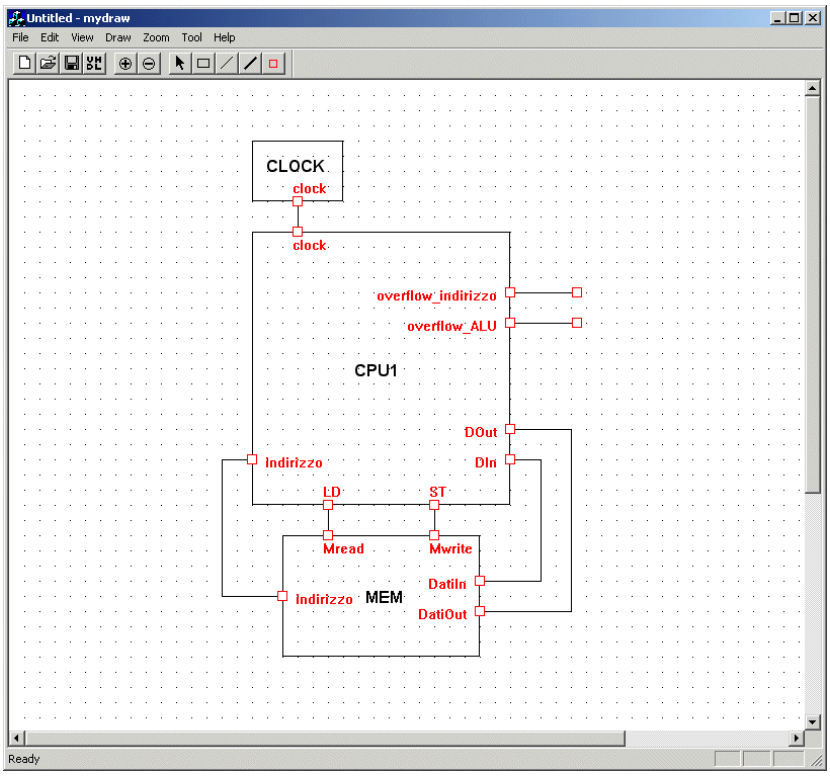


Figura 14: Test bench per la CPU1 con una sola memoria.

Si vede che vi è un solo bus su cui transitano indirizzi per le istruzioni e per i dati. Da notare i due overflow in uscita dalla CPU1: uno indica che avvenuto un overflow nel calcolo di un indirizzo di memoria mentre l'altro indica che è avvenuto

un overflow nell'eseguire una operazione aritmetica. La figura successiva rappresenta lo schema completo della CPU1 con una sola memoria che esegue le istruzioni in un solo periodo di clock.

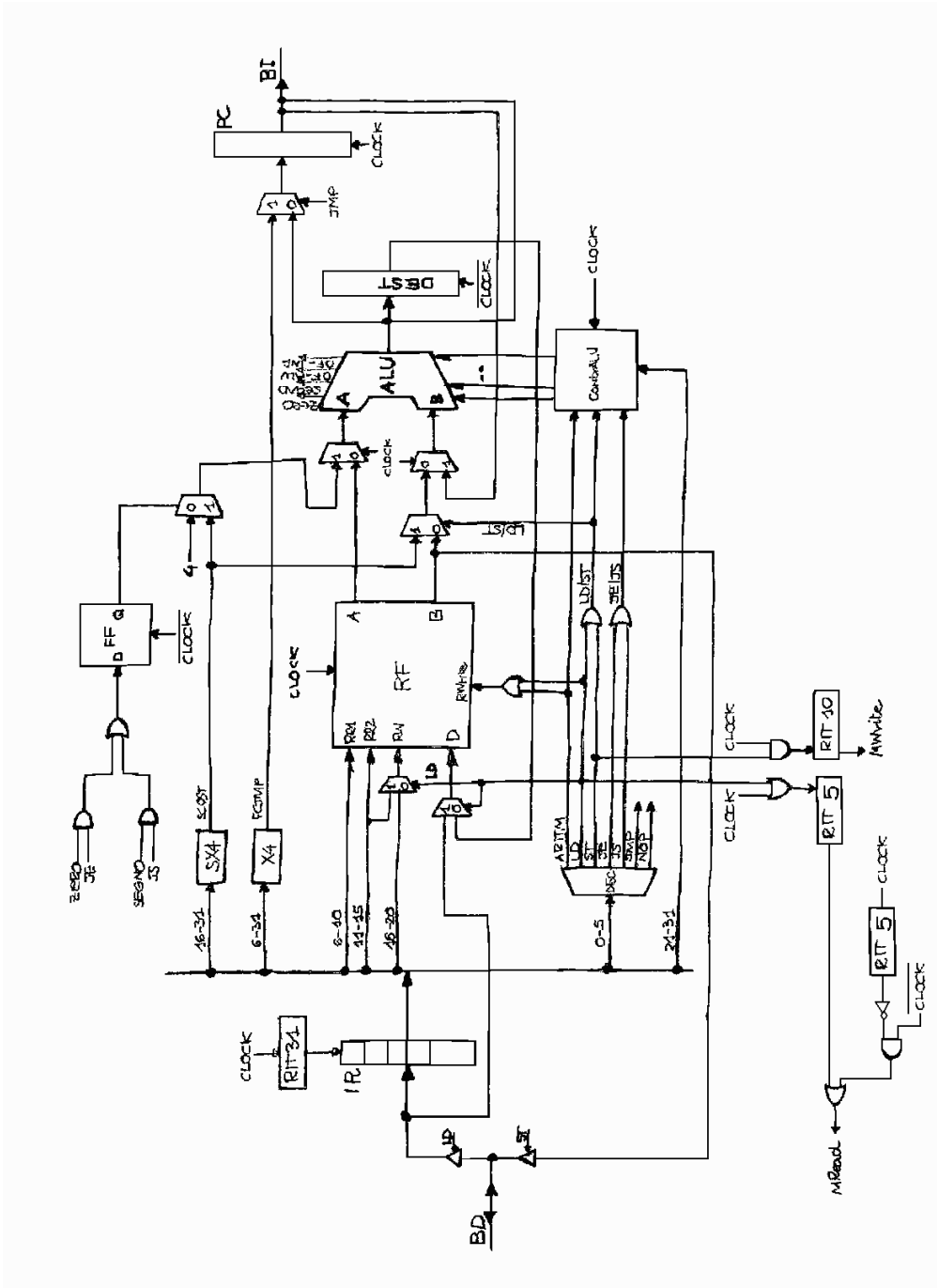


Figura 15: Schema dettagliato della CPU1 con una sola memoria.

La Figura 16 riporta un esempio di simulazione:

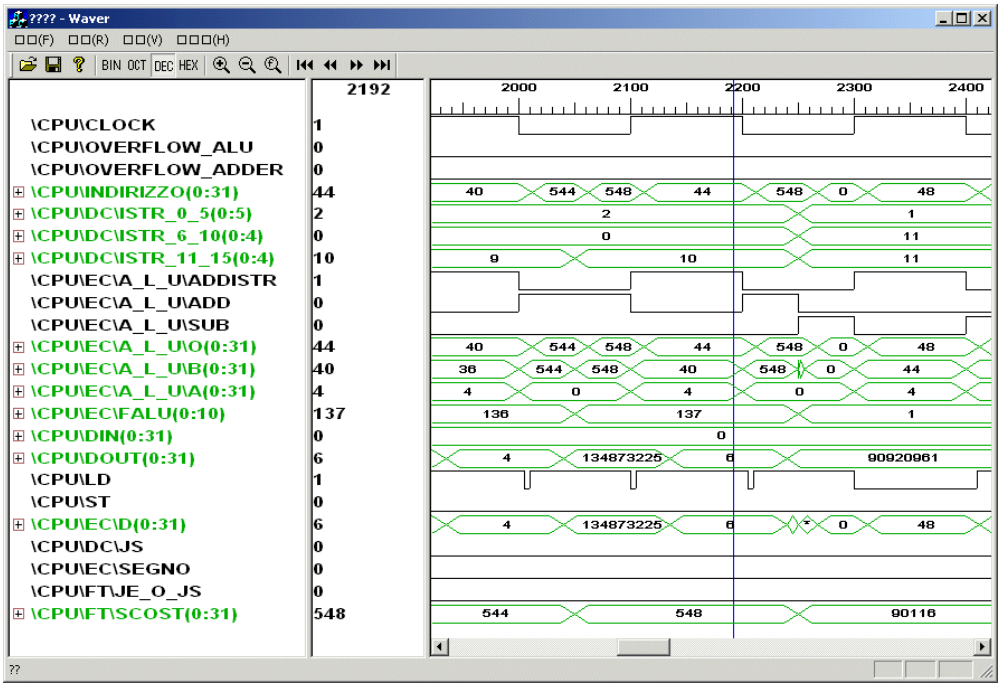


Figura 16: CPU1 con una sola memoria.

Sviluppo di CPU2

La CPU2 è un prototipo in logica cablata che esegue istruzioni in più periodi di clock. Si fa riferimento di nuovo al modello di Von Neuman, con memoria istruzioni e memoria dati indistinte. L'esecuzione di un'istruzione procede attraverso un certo numero di passi che corrispondono ad impulsi successivi del clock, cioè ai vari stati dell'automa che realizza l'unità di controllo. I cinque stati che realizzano l'unità di controllo sono la lettura dell'istruzione dalla memoria, la decodifica dell'istruzione, l'esecuzione, l'eventuale lettura o scrittura della memoria e l'eventuale aggiornamento del registro di destinazione. L'esecuzione delle istruzioni in più periodi di clock permette di unificare la memoria dati con quella istruzioni. Non è neanche più necessario il sommatore che nella CPU1 permetteva di incrementare il PC in quanto tale operazione può essere svolta dalla ALU. L'incremento del PC deve essere fatto una volta per istruzione e non ad ogni clock. Vengono inoltre aggiunti due ulteriori registri: l'IR che contiene la istruzione appena letta e un registro chiamato DEST che contiene l'indirizzo di destinazione del salto condizionato che servirà per aggiornare il PC. È poi presente una logica di controllo che genera i comandi ed i selettori con l'opportuna temporizzazione.

La CPU2 descritta negli Appunti contiene vari errori nella macchina a stati che descrive la logica di controllo e nella generazione di alcuni segnali.. Per giungere ad una versione funzionante di questo modello, abbiamo apportato varie modifiche lasciando comunque inalterato il numero di stati della logica di controllo.

4.1 Filosofia di progettazione

La realizzazione di CPU2 segue la stessa linea usata per CPU1 e riutilizza tutti i componenti che sono necessari per la sua costruzione. Si parte quindi modellando i componenti più semplici per poi collegarli per formare delle unità più complesse: questo tipo di approccio si è rivelato molto utile per individuare gli errori di modellazione dei componenti e per poter giungere ad una descrizione corretta della CPU. Infatti sarebbe stato molto difficile od addirittura impossibile tentare di descrivere il comportamento di una CPU senza prima avere a disposizione dei componenti elementari funzionanti. Anche per CPU2 si utilizzano le stesse convenzioni sui nomi dei file che sono state adottate per CPU1.

4.2 Modellazione dei principali componenti

I componenti che compongono CPU2 sono gli stessi di CPU1, con la sola aggiunta dell'unità di controllo. Le descrizioni sono tutte di tipo comportamentale ed è stato sempre realizzato un test bench per verificare il funzionamento del dispositivo. Riportiamo di seguito la descrizioni di alcuni componenti che non sono stati già descritti nella CPU1.

4.2.1 L'unità di controllo

L'unità di controllo ha il compito di generare i segnali ed i comandi per i vari dispositivi della CPU2 con l'opportuna tempificazione. Il cuore di tale dispositivo è costituito da un contatore in grado di contare fino a cinque: non è detto però che vengano utilizzati tutti e cinque gli stati. L'entità `contatore` ha due ingressi chiamati `clock` e `RSRT` che rappresentano rispettivamente il clock ed un segnale di riavvio asincrono e cinque uscite di tipo `bit` che vengono attivate in modo esclusivo e che indicano lo stato a cui siamo giunti. L'architettura di `contatore` fa uso di una variabile `stato` di tipo `integer` che memorizza lo stato del contatore. Su ogni fronte di discesa del clock la variabile `stato` viene incrementata di uno e poi viene eseguita un'operazione di modulo. Un'istruzione `case` determina quale deve essere l'uscita del contatore: solo un segnale di uscita alla volta

deve essere asserito. La logica di controllo integra questo contatore con un insieme di porte `AND` ed `OR` affinché il conteggio possa ripartire da uno prima di arrivare a cinque, in quanto non tutte le istruzioni necessitano di cinque periodi di clock per essere eseguite. Così l'entità `logica_controllo` si limita a collegare in modo opportuno il segnale `RSRT` con quelli che indicano il tipo di operazione che si sta eseguendo. La sua architettura è di tipo strutturale e contiene la dichiarazione di un componente `contatore` e di una serie di segnali che servono a collegare i vari ingressi delle operazioni con il segnale di reset del contatore. Per originare i comandi ed i selettori che controllano il comportamento delle varie unità è necessario aggiungere un'altra po' di logica che li generi a partire dallo stato in cui ci troviamo e dal tipo di operazione che si sta eseguendo: per far questo è stato creato un ulteriore componente chiamato `segnali_controllo` che genera i comandi ed i selettori in base al tipo di istruzione che è in esecuzione.

4.2.2 Un registro

Un registro è un insieme di flip flop sincronizzati da un unico clock e viene tipicamente usato come dispositivo di memoria temporanea. L'entità che modella questo dispositivo ha tre ingressi ed un'uscita. Tra gli ingressi compare il segnale `CLOCK`, un vettore di 32 `bit` chiamato `input` ed un segnale `Rin` che permette di aggiornare lo stato del registro. L'architettura dell'entità è composta da un solo processo che viene attivato ogni volta che avviene un evento sul segnale di clock; solo i fronti di discesa di clock però hanno un effetto sul componente. La variabile `contenuto` memorizza lo stato del registro; ogni volta che arriva un fronte di discesa del clock tale variabile viene aggiornata con il segnale in ingresso e poi portata in uscita.

4.2.3 Un multiplexer

Un multiplexer con due ingressi ed una uscita è un dispositivo che deve selezionare tramite un comando di selezione quale dei due segnali in ingresso deve comparire all'uscita. Poiché viene descritto in modo comportamentale, non ci interessa come è costruito al suo interno ma solo il suo comportamento esterno. L'entità che descrive un multiplexer ha in ingresso due vettori di `bit` chiamati `input_1`

ed `input_2` ed un segnale di selezione chiamato `seleziona` di tipo `bit`; in uscita vi è un vettore con lo stesso numero di `bit` degli ingressi chiamato `output`. Il processo che descrive il comportamento ha una lista di sensitività in cui compaiono i segnali di ingresso `input_1`, `input_2` e `seleziona`; una volta attivato, verifica con un'istruzione `if-then-else` lo stato del segnale `seleziona` ed assegna all'uscita `output` il valore opportuno.

4.3 Errori negli Appunti

Nella descrizione della logica di controllo della CPU2 e nella generazione di alcuni segnali riportati negli appunti, sono presenti vari errori. Dopo aver segnalato tali errori, si suggerisce una possibile soluzione.

4.3.1 Errori individuati

Lo stato 2, così come è descritto nel diagramma di stato dettagliato dell'unità di controllo e per quanto riguarda i calcoli logici dei comandi e dei selettori, è errato: la versione corretta è quella descritta a parole illustrata a pagina 22. Il quinto stato dell'istruzione load non è funzionante. Gli ingressi alla ALU per l'istruzione load riportati negli Appunti sono i seguenti:

stato 1: PC e 4 $\Rightarrow$ PC + 4;

stato 2: PC e 4 nel diagramma e PC e SCOST negli Appunti a pagina 21: la soluzione corretta è la seconda $\Rightarrow$ PC + SCOST;

stato 3: PC e SCOST $\Rightarrow$ PC + SCOST (inutile in una istruzione load/store);

stato 4: registro base e SCOST $\Rightarrow$ RB + SCOST e contemporaneamente si alzano il segnale `Mread` che genera quindi l' errore visto in precedenza di indirizzo non valido;

stato 5: alza il segnale di selezione `Sorge` in ingresso al RF che tuttavia si abbassa prima che il RF effetti la scrittura alla fine del ciclo con il risultato di scrivere un valore errato.

Le istruzioni di salto non funzionano in quanto lo stato 2 nel diagramma di stato e nei calcoli logici non è corretto. Va specificato come si genera il segnale `OPALU`

in ingresso al modulo `ContrALU`. È presente il solito errore che coinvolge la memoria dovuto al fatto che gli ingressi non sono validi quando arriva il segnale di lettura o scrittura.

4.3.2 Unità di controllo corretta

Il prototipo funzionante della CPU2 utilizza una unità di controllo a cinque stati. In riferimento allo schema di pagina 21 riportato negli Appunti, sono state apportate le seguenti modifiche:

S1:

invariato

S2:

`SALU2` = 2

S3:

JE/JS

`SALU1` = 1

`SALU2` = 1

LD (divisa da store)

`SALU1` = 1

ST

`SALU1` = 1

aggiunta di `Out` = 1

ARITM

aggiunta di `RWrite` = 1

S4:

LD

Mread = 1

ARITM

eliminazione di `RWrite` = 1

S5:

invariato

Nella pagina seguente si riporta la macchina a stati che realizza la logica di controllo con le varie correzioni apportate

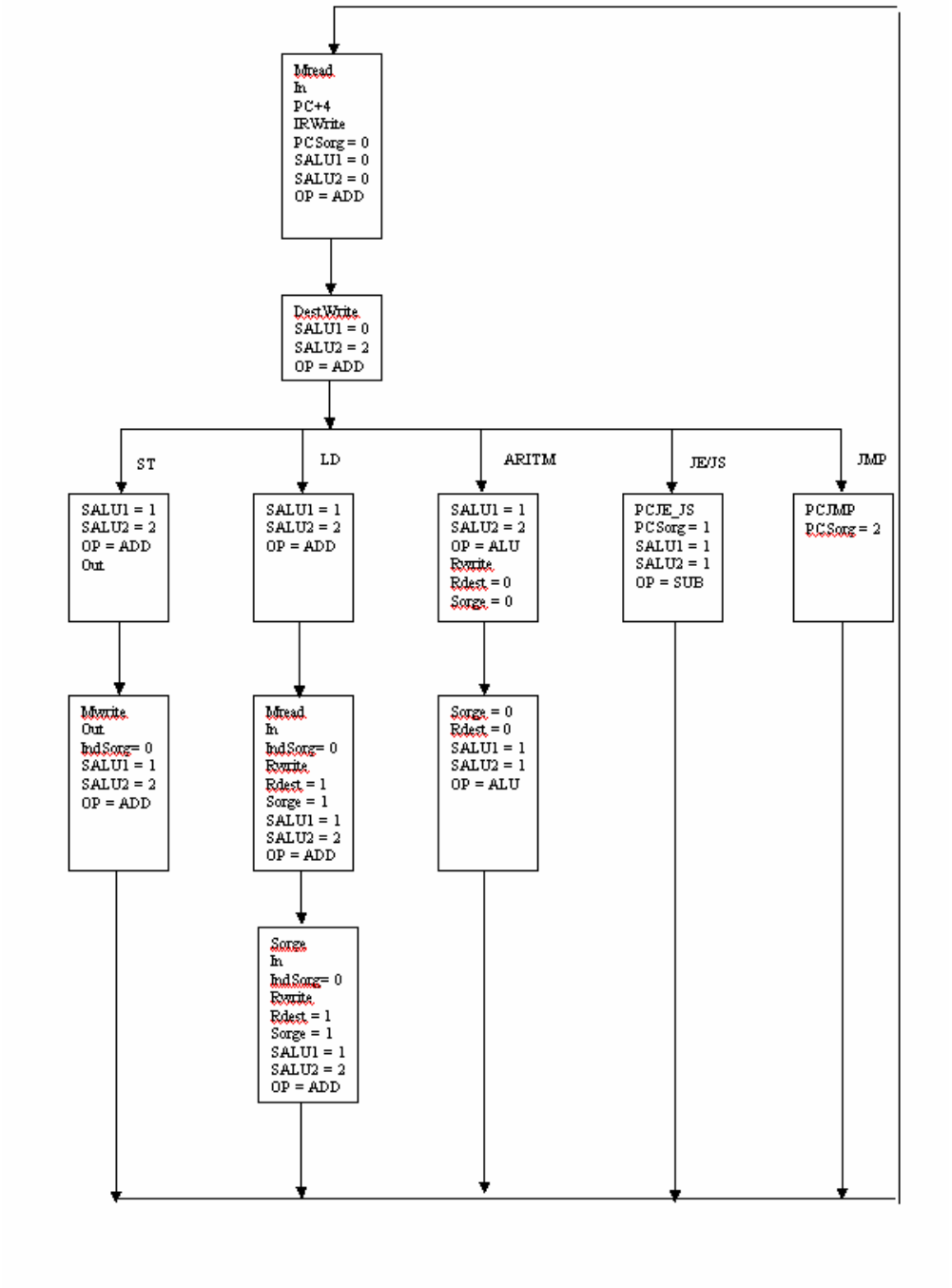


Figura 17: Diagramma di stato dettagliato dell'unità di controllo.

Oltre agli errori di lettura e scrittura in memoria già segnalati, negli Appunti compare anche un errore di scrittura sul RF. Per risolvere questo problema, si deve sfasare il segnale `RWrite` di qualche nsec, modificare la sua collocazione ed aggiungere `Sorge` = 1 nel quinto stato dell'istruzione load.

Il clock può essere fissato di durata pari al tempo di accesso della memoria con l'aggiunta di almeno un nsec di margine: per semplicità è stato fissato a 40 nsec.

4.3.3 Ritardo su alcuni segnali

Nella realizzazione di questo prototipo sono presenti delle condizioni di utilizzo di alcuni segnali proprio nel momento in cui subiscono un cambiamento di stato. Tale situazione può non venir rilevata durante una simulazione perché il VHDL deve scegliere un istante in cui prelevare il segnale. La simulazione può mascherare questo genere di errori poiché quando un processo viene attivato dalla modifica di un segnale e contemporaneamente si modificano altri segnali, quelli che non sono necessari all'attivazione del processo vengono considerati all'istante precedente. Ad esempio, se un processo viene attivato dalla modifica del segnale A e nello stesso istante si modifica anche il segnale B, nel processo attivato da A viene considerato, per ragioni di sequenziamento, il segnale B all'istante precedente. Per evitare che si verifichino tali situazioni, abbiamo analizzato attentamente il diagramma temporale prodotto dalla simulazione e ritardato quindi i seguenti segnali:

IRWrite, segnale che controlla la scrittura sul registro IR

Rwrite, segnale che controlla la scrittura sul RF

DestWrite, segnale che controlla la scrittura sul registro DEST

Mwrite, Mread, segnali di controllo della memoria

4.4 Test bench e simulazione

Per terminare la descrizione di CPU2, riportiamo il test bench ed un esempio di simulazione di tale dispositivo.

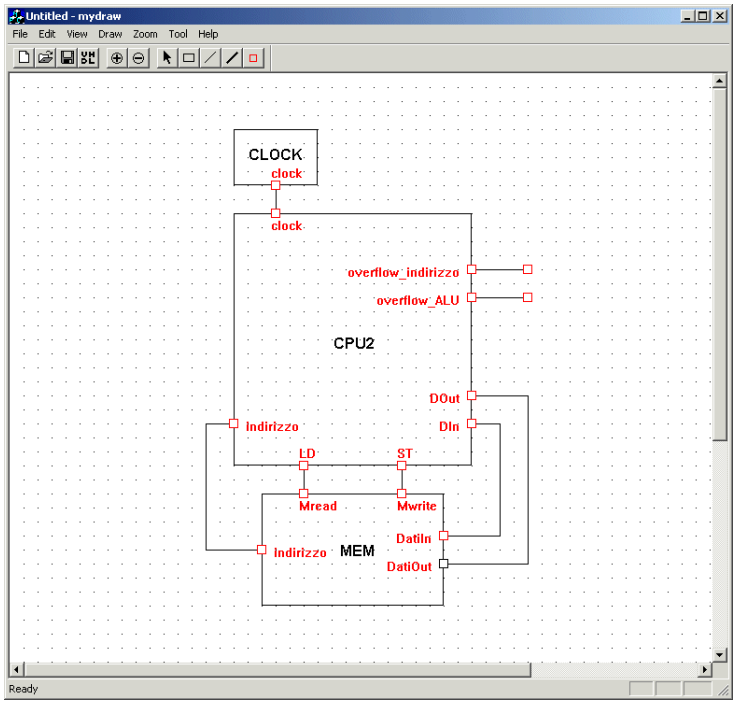


Figura 18: Test bench per la CPU2.

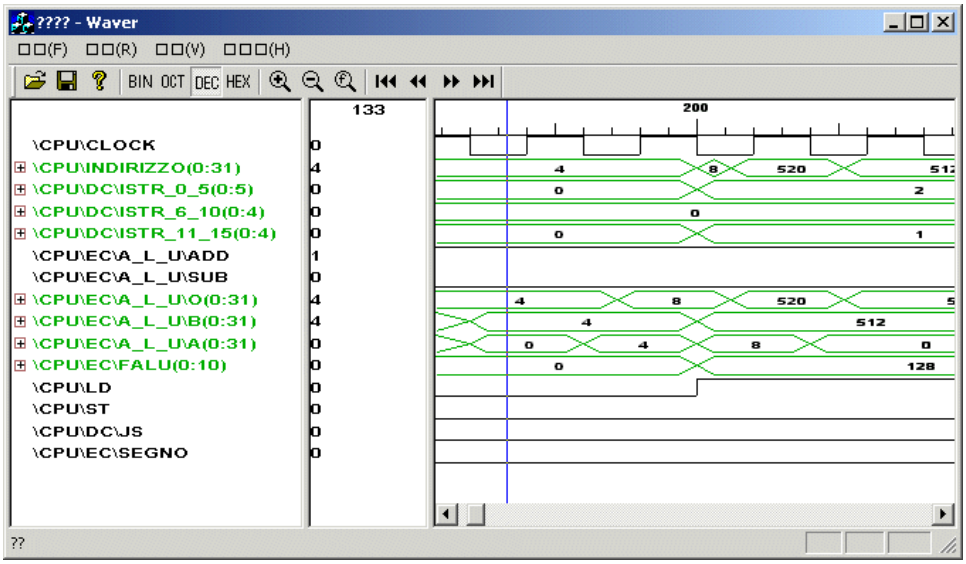


Figura 19: Simulazione di CPU2.

Un esempio di simulazione

Per simulare i componenti e le CPU realizzate in VHDL, è stato usato un simulatore chiamato Inspire. Inspire è un ambiente di simulazione che include un insieme di strumenti necessari per simulare le descrizioni VHDL, tra cui un analizzatore, un compilatore, un debugger ed un simulatore grafico. Per poter funzionare, Inspire richiede che sia installato un compilatore C/C++ e necessita dello ambiente di sviluppo Microsoft Visual Studio versione 5 o successive. Una volta installato Inspire, è necessario settare alcune variabili d'ambiente e modificare un file chiamato `.vdtrc` per fornire all'ambiente le informazioni necessarie al funzionamento. Per una descrizione dettagliata dell'installazione del programma si rimanda al tutorial fornito insieme ad Inspire. Inspire supporta una interfaccia grafica e permette di eseguire le operazioni necessarie alla simulazione in un ambiente integrato. È dotato anche di un manager che permette di richiamare i vari programmi in modo semplificato senza dover ricorre alla linea di comando.

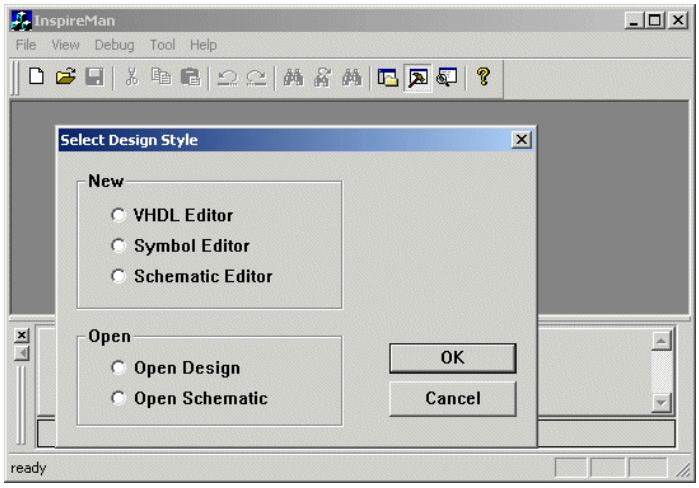


Figura 20: Il manager di Inspire.

5.1 L'assemblatore

Per poter simulare le CPU, è stato necessario scrivere un assemblatore che traducesse il codice scritto in assembler nel formato binario comprensibile per la macchina: non sarebbe stato possibile riempire la memoria in altro modo. Lo assemblatore prende in ingresso un file di testo contente il programma e fornisce in uscita un file che può essere letto direttamente dal VHDL. Con questo file è possibile riempire la parte di memoria contente le istruzioni. Ecco un esempio di file contenente del codice assembler:

```
; Questo è un commento. Esempio di file in assembler
0000-00:      nop               ; istruzione nulla
0004-01:      store 1(r0), r1   ; memorizzazione nella memoria
0008-02:      load r5, 2(r0)    ; caricamento in un registro
0010-04:      add r2, r2, r1    ; r2 := r2 + r1
0014-05:      sub r2, r2, r2    ; r2 := r2 - r2, cioè 0
0018-06:      je r2, r3, 2      ; salta a 2 se r2 = r3
001C-07:      jmp 8             ; vai a 8
```

Figura 21: Un esempio di codice assembler.

Questo piccolo programma non fa niente di sensato ma serve solo ad illustrare come deve essere scritto un file contenete del codice assembler ed il formato delle varie istruzioni. Il punto e virgola indica l'inizio di un commento che si estende fino alla fine della linea. Sulla sinistra troviamo una etichetta: può essere utile segnare i valori che indicano le locazioni di memoria occupate dall'istruzione. L'etichetta, opzionale, termina con il simbolo dei due punti. Al centro ci sono le istruzioni che compongono il programma: il formato è quello descritto nella realizzazione della CPU1. Tutti i numeri che compaiono sono esadecimali. Infine sulla destra è possibile inserire un commento libero facendolo precedere dal simbolo del punto e virgola: può essere utile inserire una illustrazione del significato dell'istruzione. Durante il processo di compilazione, l'assemblatore riporta sullo schermo, e sul file "uscita.txt", eventuali errori individuati. In uscita genera dei file in un formato di facile interpretazione per il VHDL, oltre al file "uscita.txt" che rappresenta, a compilazione corretta, il codice sorgente VHDL per quelle istruzioni.

5.2 I registri, la memoria istruzioni e dati

Per evitare di dover compilare il codice ogni volta che siano necessarie delle modifiche del contenuto iniziale dei registri, della memoria istruzioni, quando presente, e di quella dati, il simulatore legge da file il contenuto di questi componenti. Il file per la memoria istruzioni viene generato dall'assemblatore, mentre gli altri due file vengono riempiti dall'utente. Prima di lanciare il processo di simulazione, i registri, la memoria istruzioni e dati vengono riempiti con i valori letti dai rispetti file.

5.3 Il processo di simulazione

Dopo aver scritto una descrizione VHDL, il processo di simulazione si compone dei seguenti passi:

1. analisi del file contente la descrizione e generazione di un altro file contenete del codice in un formato intermedio;
2. elaborazione del codice intermedio che si compone dei seguenti sottopassi:
   2.1 generazione e compilazione di due tipi di codice, uno per la elaborazione gerarchica e l'altro per la simulazione;
   2.2 linking del codice per l'elaborazione gerarchica con il codice che costituisce il nucleo dell'elaboratore per poter realizzare un elaboratore eseguibile;
   2.3 linking del codice per la simulazione con quello che costituisce il nucleo del simulatore per generare un simulatore eseguibile;
3. apertura del simulatore eseguibile all'interno di Inspire;
4. indicazione di quali segnali devono essere stampati su schermo;

A questo punto Inspire lancia il programma che disegna l'andamento dei segnali nel tempo: il tutto avviene tramite un'interfaccia grafica. Per facilitare il processo di simulazione, Inspire mette a disposizione un manager chiamato InspireMan che integra i precedenti passi, dalla scrittura del codice VHDL fino alla simulazione dei segnali. Per quanto riguarda il primo passo, l'analisi del codice VHDL viene fatta dall'analizzatore **`VAN`**. Il secondo passo utilizza il compilatore **`IVC`** per gene-

rare i file eseguibili. Ulteriori informazioni riguardo al processo di simulazione si trovano nel tutorial incluso in Inspire. Di seguito si riportano due figure che illustrano come appaiono l'analizzatore e il compilatore all'interno dell'ambiente grafico del manager di Inspire.

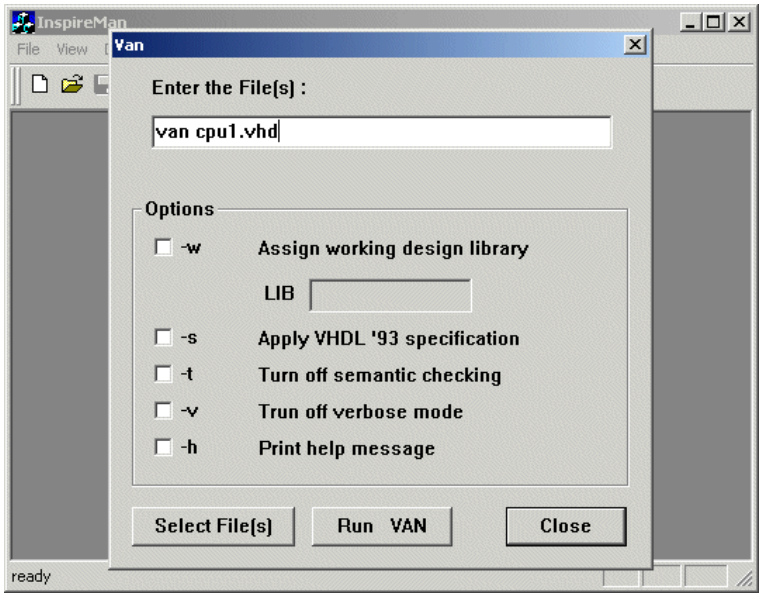


Figura 22: L'analizzatore **VAN**.

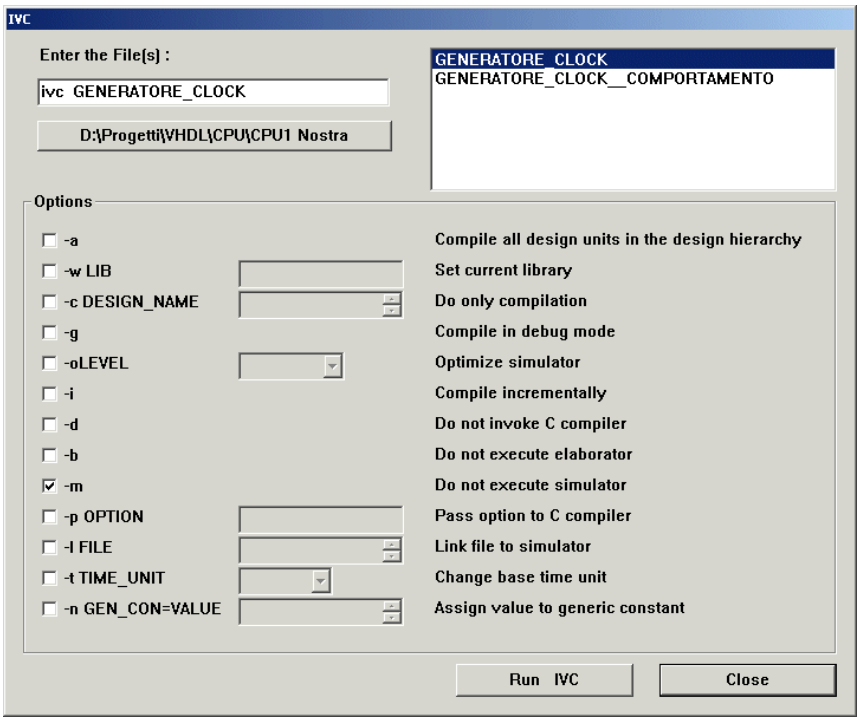


Figura 23: Il compilatore **IVC**.

5.4 Verifica del comportamento del dispositivo

Per capire se il dispositivo si comporta in modo corretto, si può analizzare il grafico prodotto dal simulatore e vedere se i vari segnali avanzano nel modo atteso. Questa soluzione è praticabile per i circuiti di bassa complessità ma risulta impraticabile per moduli complessi come una CPU. Per facilitare la verifica delle CPU realizzate, si è pensato di scrivere su file un'immagine dei registri, della memoria dati e istruzioni. Ogni elemento di memoria viene scritto su un proprio file: le scritture sono sequenziali ed avvengono ogni volta che viene attivato il processo che modella il dispositivo. Di seguito si riportano due figure che illustrano rispettivamente l'aspetto del manager prima della simulazione ed una simulazione della CPU1.

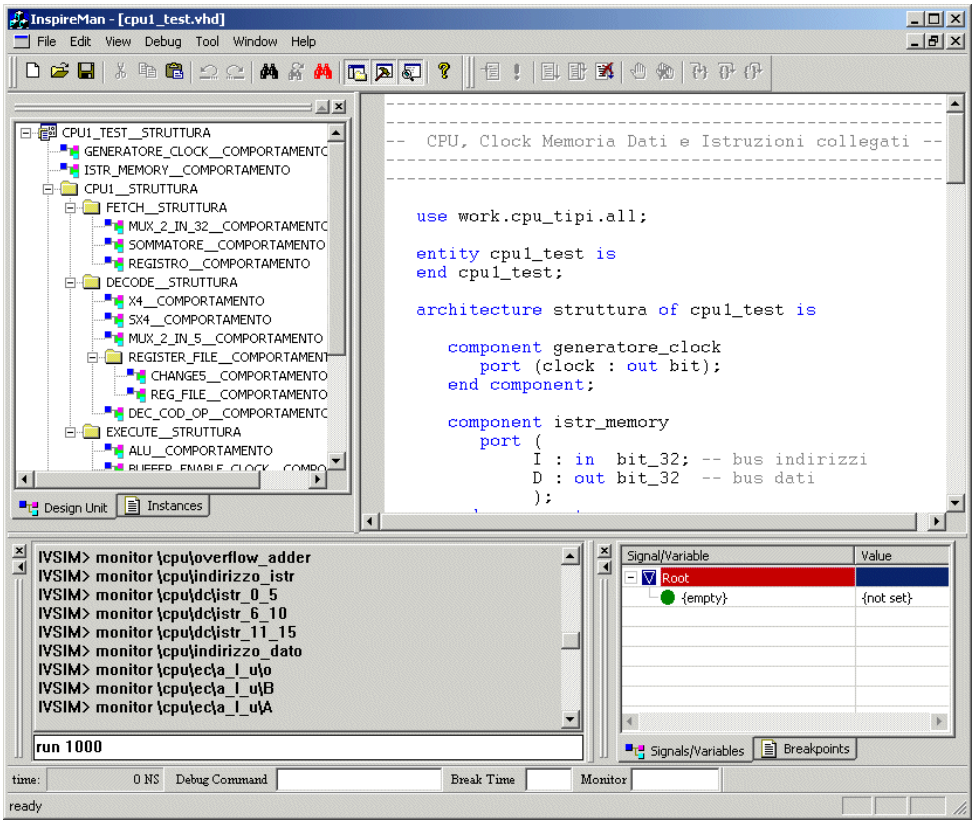


Figura 24: Il manager prima della simulazione.

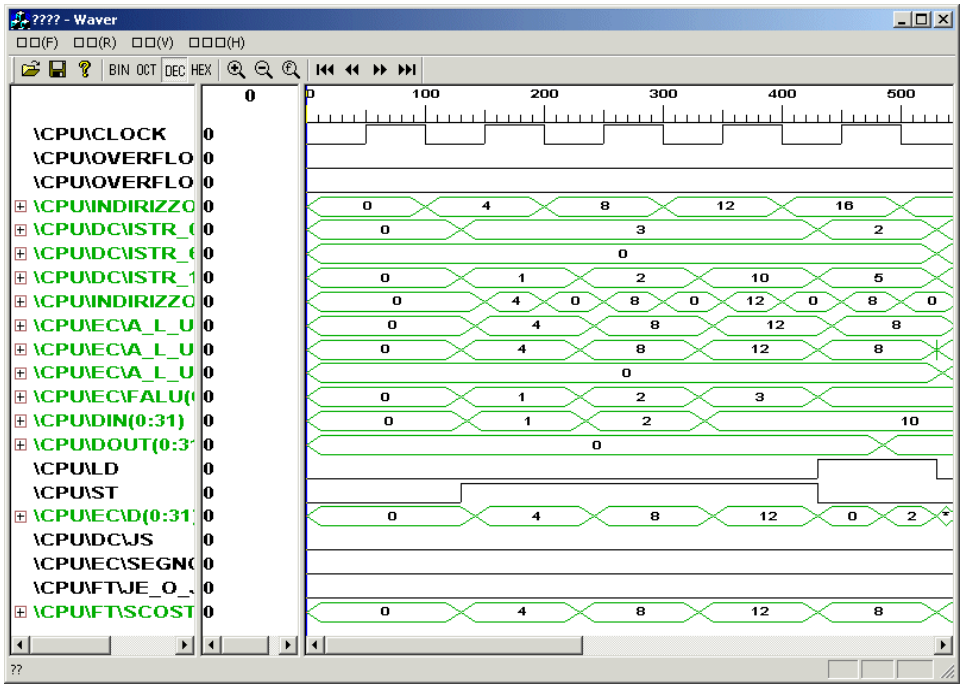


Figura 25: Un esempio di simulazione.

5.5 Il programma

Per verificare il corretto funzionamento delle varie CPU sviluppate, abbiamo scritto un programma in assembler che calcola il massimo tra dieci interi non negativi. Sono state scritte due versioni di questo programma, una per le CPU che operano con una sola memoria ed una per le CPU che lavorano con una memoria dati ed una memoria istruzioni. Le due versioni sono necessarie perché gli indirizzi cambiano passando da una realizzazione all'altra.

Indice delle figure e delle tabelle

Figura 1: Un esempio di descrizione VHDL. *4*
Figura 2: Un semplice comparatore. *6*
Figura 3: Dichiarazione di un'entità. *6*
Figura 4: Dichiarazione di un'architettura. *7*
Figura 5: L'istruzione `process`. *12*
Figura 6: Un esempio di processo. *13*
Figura 7: L'istruzione `process` *senza lista di sensitività.* *15*
Figura 8: Istruzioni concorrenti e sequenziali. *16*
Figura 9: Un esempio di test bench. *17*
Figura 10: Test bench per la CPU1 come descritta negli appunti. *23*
Figura 11: Errore della CPU1: la linea blu indica l'istante in cui avviene la scrittura. *24*
Figura 12: Generazione dei segnali `Mread` *e* `Mwrite`. *26*
Figura 13 CPU1 con le due porte AND. *26*
Figura 14: Test bench per la CPU1 con una sola memoria. *29*
Figura 15: Schema dettagliato della CPU1 con una sola memoria. *30*
Figura 16: CPU1 con una sola memoria. *31*
Figura 17: Diagramma di stato dettagliato dell'unità di controllo. *37*
Figura 18: Test bench per la CPU2. ..... *39*
Figura 19: Simulazione di CPU2. ..... *39*
Figura 20: Il manager di Inspire. *40*
Figura 21: Un esempio di codice assembler. *41*
Figura 22: L'analizzatore `VAN`. *43*
Figura 23: Il compilatore `IVC`. *43*
Figura 24: Il manager prima della simulazione. *44*
Figura 25: Un esempio di simulazione. *45*

Bibliografia

Tutto il materiale usato per poter realizzare l'elaborato è stato ricercato nel Web, oltre agli Appunti. Per quando riguarda il VHDL, le informazioni necessarie alla comprensione del linguaggio provengono da vari tutorial e da un piccolo libro introduttivo. Per quanto riguarda il simulatore, quello che ci si è sembrato il migliore tra tutti quelli visti e gratuiti è Inspire, disponibile sia per Windows sia per Linux.

[1] Peter J. Ashenden, "The VHDL Cookbook", Dept Computer Science University of Adelaide South Australia, 1990

Questo libro è scaricabile all'indirizzo:
ftp://ftp.cs.adelaide.edu.au/pub/VHDL-Cookbook/

[2] L'ambiente di simulazione Inspire è scaricabile all'indirizzo:
http://poppy.snu.ac.kr/

Altri siti contenenti tutorial, esempi e faq sono i seguenti:

[3] http://www.acc-eda.com/

[4] http://www.erc.msstate.edu/

[5] http://www.ifn.et.tu-dresden.de/

[6] http://www.vhdl.org/

[7] http://vhdl.org/

[8] http://www.shef.ac.uk/VHDL UK WWW Links.htm

[9] Vi è anche il News Group: USENET News Group comp.lang.vhdl and archive che include VHDL FAQ